TOWARDS BUILDING EFFICIENT ERROR DETECTORS FOR IMPROVING SYSTEM RESILIENCE

by

Vishal Chandra Sharma

A dissertation submitted to the faculty of
The University of Utah
in partial fulfillment of the requirements for the degree of

Doctor of Philosophy

in

Computer Science

School of Computing

The University of Utah

August 2017

The University of Utah Graduate School

STATEMENT OF DISSERTATION APPROVAL

The dissertation of **Vishal Chandra Sharma**

has been approved by the following supervisory committee members:

Ganesh Gopalakrishnan	, Chair	**10/27/16** Date Approved
Hari Sundar	, Member	**10/27/16** Date Approved
Zvonimir Rakamaric	, Member	**10/27/16** Date Approved
Vivek Srikumar	, Member	**10/27/16** Date Approved
Sriram Krishnamoorthy	, Member	**10/27/16** Date Approved

and by **Ross Whitaker** , Chair/Dean of

the Department/College/School of **Computing**

and by David B. Kieda, Dean of The Graduate School.

ABSTRACT

Current scaling trends in transistor technology, in pursuit of larger component counts and improving power efficiency, are making the hardware increasingly less reliable. Due to extreme transistor miniaturization, it is becoming easier to flip a bit stored in memory elements built using these transistors. Given that soft errors can cause transient bit-flips in memory elements, caused due to alpha particles and cosmic rays striking those elements, soft errors have become one of the major impediments in system resilience as we move towards exascale computing.

Soft errors escaping the hardware-layer may silently corrupt the runtime application data of a program, causing silent data corruption in the output. Also, given that soft errors are transient in nature, it is notoriously hard to trace back their origins. Therefore, techniques to enhance system resilience hinge on the availability of efficient error detectors that have high detection rates, low false positive rates, and lower computational overhead. It is equally important to have a flexible infrastructure capable of simulating realistic soft error models to promote an effective evaluation of newly developed error detectors.

In this work, we present a set of techniques for efficiently detecting soft errors affecting control-flow, data, and structured address computations in an application. We evaluate the efficacy of the proposed techniques by evaluating them on a collection of benchmarks through fault-injection driven studies. As an important requirement, we also introduce two new LLVM-based fault injectors, KULFI and VULFI, which are geared towards scalar and vector architectures, respectively. Through this work, we aim to make contributions to the system resilience community by making our research tools (in the form of error detectors and fault injectors) publicly available.

CONTENTS

LIST OF TABLES

LIST OF FIGURES

ACKNOWLEDGMENTS

My journey as a Ph.D. student has been full of challenges, but it has also been intellectually fulfilling and a rewarding experience. I would like to acknowledge those people who have very generously helped me during various stages of my student life here at Utah.

First and foremost, I would like to thank my advisor Prof. Ganesh Gopalakrishnan who has been instrumental in shaping my whole Ph.D. experience. I am greatly thankful for his unwavering support which includes research advice, financial support, and giving me pep talks whenever I needed them. I am equally grateful to him for being always very kind and generous in finding time for me out of his amazingly busy schedule to listen to my research ideas, providing honest feedback, ignoring my naïvety at times, and teaching me to write technical papers. Next, I would like to thank my co-advisor Prof. Hari Sundar who very kindly agreed to advise me during the most recent part of my Ph.D. work, and has made a strong impression in this relatively small period through his timely advice and sharp feedback.

I would also like to very sincerely thank the rest of my committee members starting from Prof. Zvonimir Rakamarić who provided me excellent guidance and nurtured my interest in compilers during the initial part of my Ph.D. work. Next, I am thankful to Prof. Vivek Srikumar for being always generous with his time, and for his willingness to help me whenever I am in need. My particular thanks to Dr. Sriram Krishnamoorthy for always patiently listening to and brainstorming various research ideas on numerous occasions, and helping in carefully choosing the better ones. His sharp feedback and great research and writing skills always pushed me to keep on improving our work. I would also like to thank Dr. Greg Bronevetsky for his support and guidance during one element of my initial research work. Thanks to all my friends in Gauss research group and academic advisors Ann and Leslie for their support and help. Finally, this list cannot be complete without thanking my wife Rashmi, my parents, and my siblings for their sacrifices and support.

CHAPTER 1

INTRODUCTION

High-performance computing (HPC) applications will soon be running at very high scales on systems with large component counts. The shrinking dimensions and reducing power requirements of the transistors used in the memory elements of these massively parallel systems make them increasingly vulnerable to temporary bit-flips induced by system noise or high-energy particle strikes. The transient bit-flips occurring in memory elements are often referred to as soft errors (also called *single-event-upsets*). Soft errors are one of the gravest of impediments to the rapid attainment of large-scale (especially exascale) computing capabilities [65,86,89]. Such errors are typically caused by radiation from chip packaging, cosmic rays [45,91], or even circuit noise due to low-power operation [15].

May et al. reported one of the earliest accounts about the presence of soft errors in 1979 when they witnessed single-bit error in dynamic random-access memory (DRAM) due to alpha particles [62]. They attributed the alpha particle strikes to radioactive contamination in packaging materials. A much later study done by Michalak et al. [65] reported evidence of soft errors affecting ASC Q supercomputer's onboard static random-access memory (SRAM) cache lines. The onboard cache was not protected using error-correcting code (ECC), and was only capable of detecting errors with an odd number of bits by using an additional parity bit per cache line. The authors were able to validate their hypothesis that the errors reported by the parity-bit detectors were due to soft errors caused by cosmic rays.

Recent studies project an upward trend in soft-error-induced vulnerabilities in HPC systems, thereby pushing down their mean time to failure (MTTF) [18,19]. These studies have also highlighted that soft-error-rate (SER) in storage elements such as central processing unit (CPU) register files, cache memory, and DRAM are expected to rise in the future due to aggressive feature scaling and constant demand for power-efficient architectures. These trends drastically increase the likelihood of a bit-flip occurring in long-lived

computations. A bit-flip affecting computational states of a program under execution such as arithmetic and logical unit (ALU) operations or live register values may lead to a silent data corruption (SDC) in the final program output. Making the matter worse, such erroneous values may propagate to multiple compute nodes in massively parallel HPC systems [6]. Also, due to the transient nature of soft errors, they are notoriously difficult to trace back and it is equally difficult to precisely predict their occurrence rate for a given hardware component [67]. Therefore, researchers are actively working on finding new ways to accurately estimate soft-error-rate (SER) [56,71]. These facts highlight the need for developing proactive strategies to provide sufficient coverage for detecting and correcting soft errors.

1.1 A Cross-Layer Approach to System Resilience

Apparently, soft errors have already been identified as one of the key challenges to system resilience as we transition to exascale computing. While additional or hardened hardware circuitry can prevent the onset of failures or detect them early in the stack, with the ever increasing push for power-efficient architectures, cross-layer resilience solutions (as highlighted in Figure 1.1) that encompass all of the hardware and software stacks, are

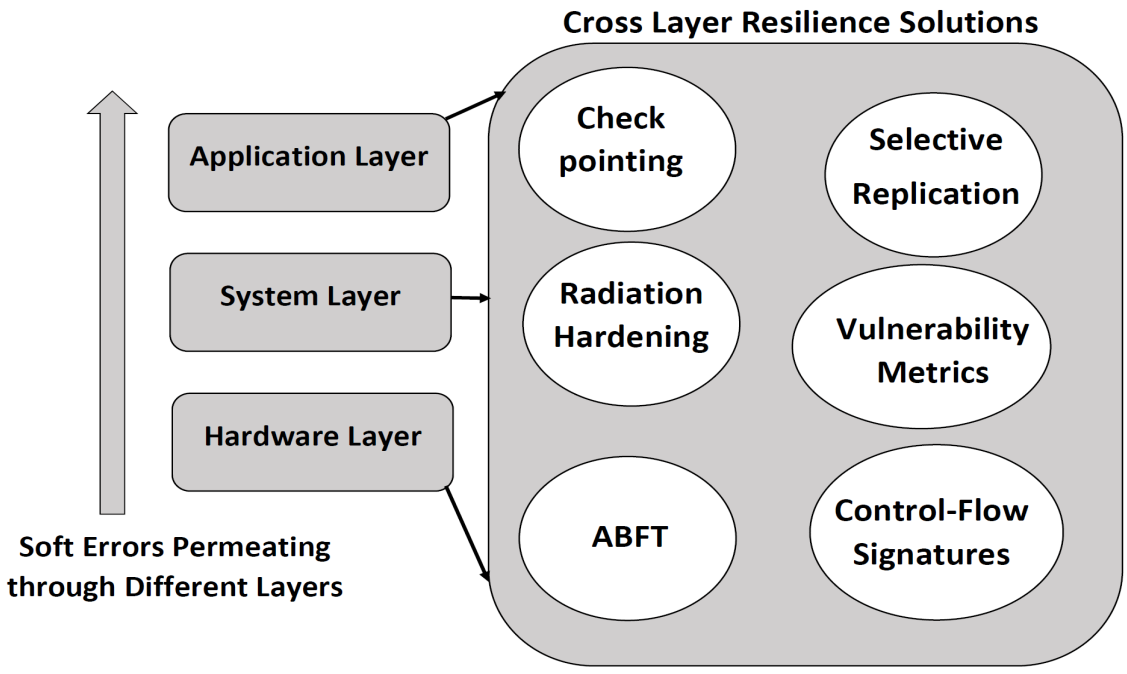

Figure 1.1: A cross-layer approach to system resilience

essential to be developed [18,19,25].

Given that resilience solutions typically incur a net drop in performance, it is critical that the research community develops and makes available a whole array of solutions that can then be evaluated over a sufficient period by practitioners, so that the right combinations of solutions that minimize overheads may be chosen. Studies [87] indicate that efficient software-level error detectors are necessary to reduce the use of the 'always on' hardware-level checkers that can exacerbate energy consumption.

We now list some of the key resilience solutions which are employed across different layers of a system.

1. **Hardware-level techniques**: Selective hardening techniques such as Triple Modulo Redundancy (TMR), SECDED, and Chip-Kill are employed to protect critical hardware elements. Identification of critical elements is made using vulnerability analysis such as Architecture Vulnerability Factor (AVF) proposed by Mukherjee et al. [68]. Hardware-only protection mechanisms are cost sensitive and are often limited by budgetary constraints. Therefore, cross-layer resilience solutions are envisaged as the way forward.
2. **Compiler-level techniques**: Compiler-level techniques often involve selective duplication of instructions. Considerable research has been done at the compiler layer to automatically identify vulnerable code regions which are then used to guide placement of the detectors [38,60,75,77,92].
3. **Checkpointing**: The method of checkpointing requires saving application states at regular intervals to an array of disks such as redundant array of independent disks (RAID). In the event of failures, the checkpointed data is used to restore the execution context to the last known good condition. The frequency of checkpointing is often guided by the estimated MTTF of the underlying system [17,19]. Due to the associated performance and space overhead with this approach, determining an optimal frequency for checkpointing and reducing the size of the checkpointing data are the actively researched areas.
4. **Algorithm-based Fault Tolerance (ABFT)**: Techniques falling under this category exploit algorithmic properties of a program to detect the presence of soft errors during its execution, for example, maintaining checksum of rows and columns data of matrices used in a matrix-matrix multiplication [34] and using the checksum to

detect single-bit errors.

5. **Control-Flow Protection**: Soft errors inducing control-flow deviations may cause SDC in a program's final output. Therefore, a separate line of research has focused on extracting efficient control-flow signatures to detect illegal control-flow deviations caused due to soft errors [14, 50, 72, 80, 96, 106, 107].

1.2 Evaluation of Resilience Solutions

Developing a robust evaluation infrastructure capable of simulating realistic error models is an essential requirement for evaluating any resilience solution. For example, a recent study by Cher et al. uses hardware-level proton irradiation as well as software-level fault injections to assess fault tolerance capability of BlueGene/Q supercomputers against soft errors [26]. Similarly, Fang et al. have developed an LLVM-level fault injector and they present a fault injection driven study to assess resiliency of OpenMP programs against soft errors [36]. Another study, done by Hari et al., involves performing fault injections using a microarchitectural simulator to find those application fault sites which lead to silent data corruption. One of the key focus of their study is to find a minimal set of representative fault sites which can be used to characterize the resiliency of an application [43]. The underlying importance of a flexible and efficient fault injection framework lead our development efforts towards designing fault injectors for scalar and vector architectures.

1.3 Thesis Contributions and Organization

1.3.1 Fault Injectors for Scalar and Vector Architectures

The previous section highlighted the need for having a flexible and effective evaluation infrastructure. To this goal, we have developed two instruction-level fault injectors, KULFI and VULFI, which are targeted for scalar and vector architectures, respectively [95, 96]. Chapter 2 details the design and workflow of these error injectors including their fault site selection and fault injection algorithms.

1.3.2 Control-Flow Detectors Using Predicate-Abstraction

The next contribution of this dissertation work (described in Chapter 3) is a novel control-flow error detector capable of detecting control-flow deviations induced by bit-flips [96]. Using a well-known formal technique called predication-abstraction [8, 28, 42],

our technique identifies a set of valid predicate transitions through dynamic profiling which are then used as error detectors.

1.3.3 Automated Synthesis of Predicate Transitions-based Detectors

Chapter 3 also introduces our third key contribution in the form of a tool called FUSED which automates the process of profiling predicate transitions in a target program and inserting the error detectors, built using these transitions, back into the target program [98].

1.3.4 Control-Flow Detectors for Vector Loops

Chapter 3 also describes in detail our fourth contribution as another set of control-flow detectors targeting vector loops [95] generated by Intel's ISPC compiler [3,81]. This work introduces a novel approach for analyzing code generation strategy adopted by the ISPC compiler to infer compiler-level invariants which are then used for error detection.

1.3.5 Building Error Detectors for Stencil Computations

Our earlier work on control-flow detectors raised our curiosity to explore the feasibility of lightweight detectors for applications which are not rich in control-flow. Chapter 4 presents an exploratory work which presents a feasibility study on building low overhead detectors for stencil computations [93]. Stencil computations usually have high arithmetic intensity with a low count of control predicates. We present a systematic approach to extract a lightweight approximate kernel for a target stencil kernel using machine learning, run the approximate kernel alongside the main kernel, and compare their results to detect errors. Chapter 4 shares the finding of the feasibility study and makes key recommendations for building effective detectors for stencil computations.

1.3.6 Protecting Structured Address Computations

Our last key contribution (also our most recent work), presented in Chapter 5, proposes a compiler-level technique to protect structured address computations [94]. Unfortunately, efficient detectors to detect faults during address generation (to index large arrays) have not been widely researched. We present a novel lightweight compiler-driven technique called PRESAGE for detecting bit-flips affecting structured address computations. A key insight underlying PRESAGE is that any address computation scheme that flows an already incurred error is better than a scheme that corrupts one particular array access

but otherwise (falsely) appears to compute perfectly. Flowing errors allows one to situate detectors at loop exit-points, and helps turn silent corruptions into easily detectable error situations. Our experiments using PolyBench benchmark suite indicate that PRESAGE-based error detectors have a high error-detection rate and incur low overheads.

1.4 Thesis Statement

The previous section summarizes our unique contributions which revolve around analyzing a program's properties such as control predicates, code generation pattern in the case of vector compilers, the structure of a stencil computation for learning approximate kernels, and structured address computation logic at compiler-level, to build efficient error detectors. The evaluation of these techniques is possible due to the fault injectors developed as part of work. Therefore, we describe our thesis statement as follows:

> Analyzing program properties at the application and intermediate representation levels to develop lightweight and efficient error detection techniques, and evaluating the efficacy of these techniques using a flexible and robust fault injection infrastructure driven towards improving system resilience.

1.5 Multiple Authors Release Statement

The contributions listed above are based on the publications which I have coauthored with one or more people from the following list: Ganesh Gopalakrishnan, Zvonimir Rakamarić, Sriram Krishnamoorthy, Greg Bronevetsky, and Arvind Haran. Duly filled multiple-author-release forms signed by the coauthors listed above, granting me the permission to use the content of our joint publications in my dissertation, will also be submitted along with this dissertation work. I would like to sincerely thank the coauthors for giving their consent to use the content of our jointly published work towards my dissertation.

CHAPTER 2

BUILDING A FLEXIBLE FRAMEWORK FOR SOFT ERROR SIMULATIONS

2.1 Introduction

Having a flexible and robust infrastructure capable of simulating various soft error scenarios is a necessary and essential prerequisite for evaluating any new resilience solution. Unfortunately, we could not find any publicly available compiler-level fault injector when we initially started our resilience research. Therefore, to this goal, we developed two instruction-level fault injectors named KULFI and VULFI. Both of these injectors are developed using LLVM compiler infrastructure [55, 58] and they perform fault injections at LLVM intermediate representation (IR) level. Our initial effort led to the development of KULFI (Kontrollable Utah's LLVM-based Fault Injector) which supports injecting faults in LLVM instructions which are of *scalar* type. As the focus of our research shifted more towards HPC domain where programs are often subjected to vectorization, we developed VULFI (Vectorized Utah's LLVM-based Fault Injector) to handle vector instructions. Specifically, VULFI is capable of targeting both vector and scalar instructions thereby subsuming KULFI, and currently supports languages such as C, C++, ISPC, and OpenCL. Moreover, VULFI's modular software design makes it much easier to implement new error models. This chapter explains salient features of both KULFI and VULFI tools, detailing their designs, workflows, and demonstrating their effectiveness through experimental evaluations.

2.2 Error Models

On a broader level, the various error models considered in this dissertation work share a common error scenario where soft errors induce a single-bit fault affecting CPU register files and ALU operations. We assume DRAM and cache memory to be error-free, which

is a reasonable assumption as they are often protected using ECC mechanism [23, 51, 52, 100]. The aforementioned error scenario is implemented by targeting runtime instances of LLVM IR-level instructions of a target program for fault injections. For example, if there are N dynamic IR-level instructions observed corresponding to a target program, then we choose one out of N dynamic instructions with a uniform random probability of $\frac{1}{N}$. After that, we flip the value of a randomly selected bit of the destination virtual register, i.e., the l.h.s. of the randomly chosen dynamic instruction. We call each dynamic instruction a dynamic fault site, and the respective static instruction a static fault site. Both KULFI and VULFI support various fault site selection strategies, e.g., selecting only those fault sites which pertain to pointer arithmetic.

2.3 Classifying Outcomes of Fault Injections

Classifying the result of a fault injection performed on a target program requires comparing the output produced by the *faulty* execution with that of a fault-free execution such that both runs are carried out using the same set of test inputs. During the faulty execution, a single-bit fault is injected into a randomly chosen dynamic instruction as explained earlier. Both KULFI and VULFI classify the outcome of the faulty execution into one of the following categories by comparing its output with that of the fault-free execution:

1. **Silent Data Corruption (SDC)** : When the result of the faulty execution differs from that of the fault-free execution.
2. **Benign** : When no difference is observed between the executions.
3. **Program-crash/Segmentation Fault** : When the faulty execution results in a system failure, a program-crash, or any other issue that could easily be detected by the end user.

2.4 KULFI: A Fault Injector Targeting Scalar Instructions

KULFI is capable of injecting dynamic faults into programs written in C. A dynamic fault emulates a transient fault induced by soft errors, and is injected into a fault site selected during program execution. KULFI injects a single-bit fault into a data or a pointer register by flipping a randomly chosen bit in the live register value. It provides fine-grained control over the fault injection process by allowing a user to specify fault injection

probability, injected byte location, fault site type (data, pointer), etc. Figure 2.1 shows the flowchart of the dynamic fault injection done by KULFI. At a high level, the fault injection loop goes through all dynamic instructions. For each dynamic instruction, the fault type to be injected is selected as either data or pointer type. Subsequently, KULFI checks whether it is feasible to inject a fault with the chosen fault type into the selected instruction. If this test passes and the provided fault injection probability is met, then a fault is injected into the instruction. We repeat these steps for all dynamic instructions. Once the loop is finished going through all the dynamic instructions, the execution of KULFI terminates. Figure 2.2 illustrates a transient fault occurring at register level. The shown register does not contain a fault at time t_1. At time t_2, a fault occurs, and then it disappears at time t_3. Dynamic fault injection capability of KULFI models such transient fault behavior. KULFI operates on the LLVM intermediate representation (IR) level (i.e., LLVM bitcode level) in the static single assignment (SSA) form. SSA ensures that every IR variable (i.e., logical register) is assigned only once, which is an advantage as opposed to operating at the source code level when modeling transient faults. More specifically, injecting a fault into an SSA logical register referenced by an instruction is a one-time occurrence affecting only the instructions that use that logical register. SSA naturally prevents references to the same source code variable in the later instructions from observing the injected fault. Note that the duration for which a transient fault persists in an actual hardware register varies. Therefore, it is possible that more than one instruction could get affected by a single transient fault. Currently, we do not capture such timing-related behaviors of transient faults in the software emulation of faults done by KULFI. However, KULFI still provides a reasonable model of transient fault behaviors, and similar models were adopted by other error injectors [31,59].

2.5 Evaluation of KULFI

For evaluating KULFI, we carry out an empirical case study that assesses the resilience of several popular sorting algorithms. Previous work suggests that algorithms and data structures that solve a particular problem not only vary in time and space complexity, but also in how resilient they are to faults [40]. In that line of work, a memory (i.e., DRAM) error model is assumed to study the resilience of sorting algorithms [39]. However, the memory error model is often too restrictive since it fails to cover classes of faults not

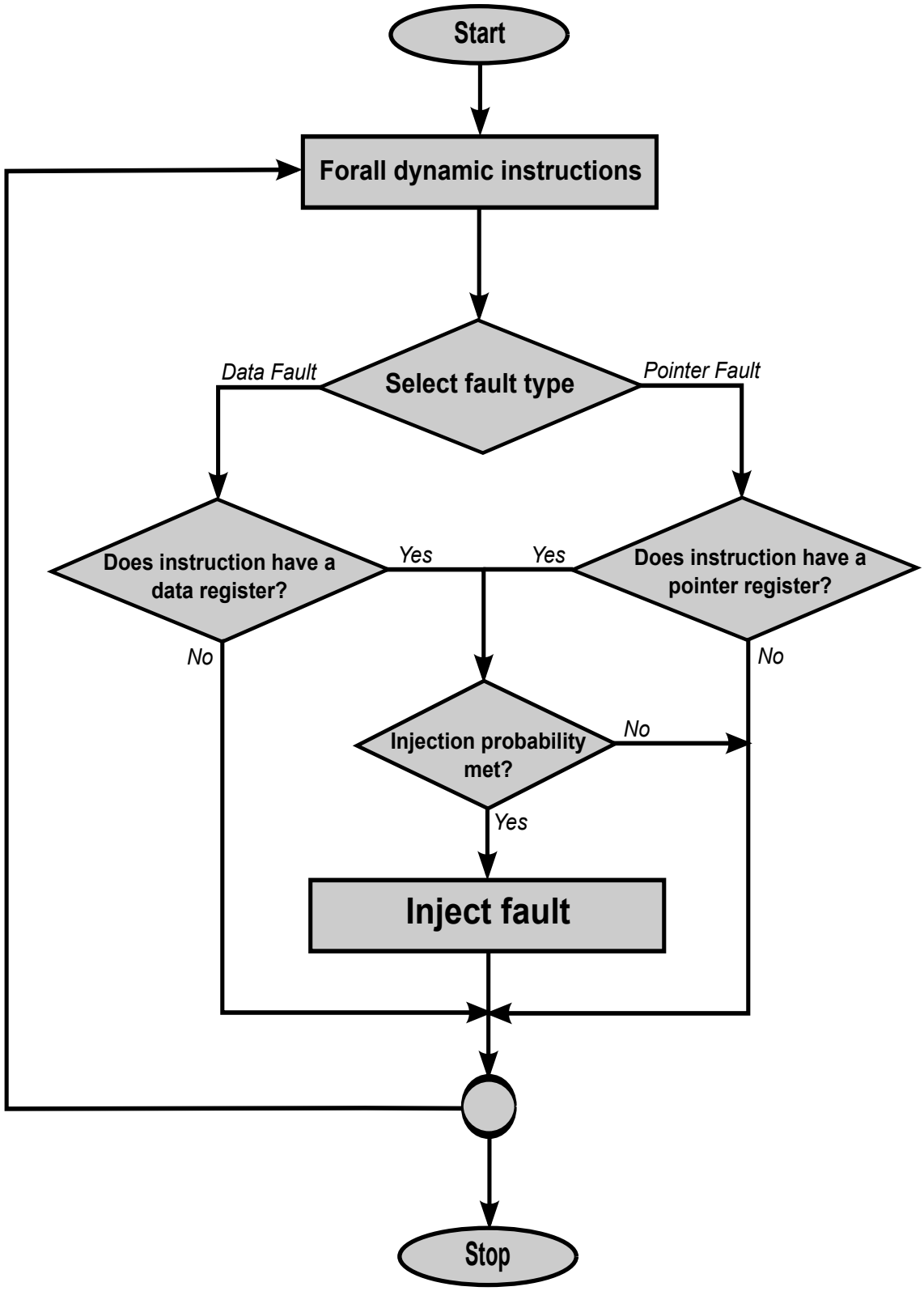

Figure 2.1: Flowchart showing dynamic fault injection in KULFI

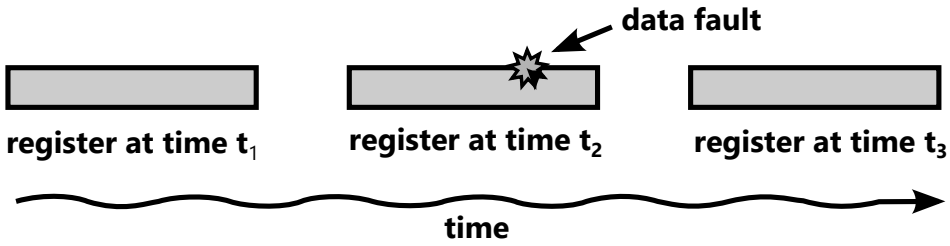

Figure 2.2: A transient fault occurring in a register

directly tied to memory, such as register corruptions, control-flow corruptions, and incorrect computation, which are prevalent in real-world systems. In our empirical study, we choose to use a more descriptive error model supported in KULFI. The selected error model considers all instructions of a program as candidate fault occurrence locations, including memory reads and writes, register operations, and control-flow instructions.

In our case study, we consider implementations of five well-known sorting algorithms: BubbleSort (with preemptive termination criterion), RadixSort, QuickSort, MergeSort, and HeapSort.[1] All implementations take as input an array of integers to be sorted, and they output the sorted array. In this study, we do not bias on the size and input data, i.e., the arrays are of arbitrary size (between 2000 and 10000) and contain random integer data. We perform a *fault injection campaign* for each sorting algorithm implementation using KULFI. Each fault injection campaign consists of 200 *fault injection experiments*. A single fault injection experiment comprises 100 executions of an algorithm. Therefore, each algorithm is executed a total of 20000 times, which we split into 200 fault injection experiments so that we can later compute the statistical significance of our results. In each execution, the algorithm operates on a different randomly generated input array, while a single random bit-flip error is injected at runtime using KULFI. We describe the details of our fault injection strategy next.

2.5.1 Execution Strategy

Even with a fault injector such as KULFI available, selecting a realistic fault injection probability requires careful planning; we now present our approach in this regard. Fault filtration naturally occurs across the hardware/software stack where many faults fall into

[1]Source code of the examples and scripts for performing the experiments are also available from the KULFI website.

the ”don’t-care” sets of the higher layers. Specifically, Sanda et al. [91] report how an IBM POWER6 processor was bombarded with protons and alpha particles within an elaborate experimental setup. The authors estimated that the percentage of faults that reached the application logic was 0.2% of the overall number of latch-level faults. While this approach to fault simulation is quite realistic, such a ”bottom-up” fault injection approach (and its infrastructural overheads) is clearly out of reach for most researchers. On the other hand, there are some recent approaches targeting software-level resilience-enhancing mechanisms Therefore, we decided to focus our empirical study only on the effects of faults that *do reach* the application logic since those are of particular interest to the software-level resilience community. We still had to devise a reasonable and fair fault injection probability.

Given the above discussion, we now define additional notions that help us elaborate our studies. By the term *dynamic instruction*, we refer to a runtime instance of a static LLVM program instruction. We define the *dynamic instruction count* as the actual number of dynamic LLVM instructions executed corresponding to a specific program execution. For example, for a simple program consisting of five static instructions in a loop that iterates 1000 times, the static instruction count is five, while the dynamic instruction count is 5000. For our sorting algorithms, the dynamic instruction count varies depending on the algorithm considered and the input array, which we have to take into account to ensure that all dynamic instructions are considered for fault injection with equal probability. Table 2.1 gives various statistics for our sorting algorithms:

- LOC is the number of lines of code,
- SIC the number of static fault site instructions,
- MinDIC the minimum dynamic instruction count,
- MaxDIC the maximum dynamic instruction count, and
- AvgDIC the average dynamic instruction count.

Table 2.1: Experimental statistics of sorting algorithms

Algorithm	LOC	SIC	MinDIC	MaxDIC	AvgDIC
BubbleSort	56	13	68k	61442k	14818k
RadixSort	61	39	30k	2040k	565k
QuickSort	65	25	34k	1110k	303k
MergeSort	70	38	79k	1269k	364k
HeapSort	77	28	15k	1519k	500k

We initially perform a fault-free execution of an algorithm on an input to compute the dynamic instruction count N for a particular execution. We then define the probability of fault injection for each dynamic instruction to be $1/N$. This ensures that all dynamic instructions are equiprobable for fault injection in subsequent runs of the program on the same input.

Figure 2.3 illustrates our fault injection strategy for this case study performed using KULFI. First, a sorting routine is compiled using LLVM's C front-end Clang into an LLVM bitcode file, which contains LLVM's intermediate representation. Then, we execute the generated bitcode file using the LLVM virtual machine (i.e., *lli*) and as input, we provide a randomly generated input array. We record the sorted output array for later comparison. In the process, we also measure the dynamic instruction count N for this particular fault-free execution. Using the dynamic instruction count, we compute the probability of fault injection for each dynamic instruction as $1/N$.

The original LLVM bitcode file and the computed fault injection probability are given as inputs to KULFI. The tool generates a fault-injecting LLVM bitcode file, i.e., an instrumented version of the original bitcode file in which a transient fault might be injected during execution into a dynamic instruction with the computed probability. The fault-injecting LLVM bitcode file is then executed on the same input array. We observe the number of injected faults and log only the executions during which exactly one fault is injected; we call such executions 1-fault executions. Executions, where the number of injected faults is not equal to one, are discarded. We record the outcome of every 1-fault execution to analyze the effect of fault injection later.

2.5.2 Experimental Results

After each fault injection experiment (i.e., 100 1-fault executions), we log the number (i.e., fraction) of executions falling into each category. For example, here is how one such log entry might look like:

```
Benign: 41, Segmentation: 29, SDC: 30
```

At the end of every sorting algorithm fault injection campaign, we are left with 200 such log entries, one for every fault injection experiment. We perform statistical analysis of

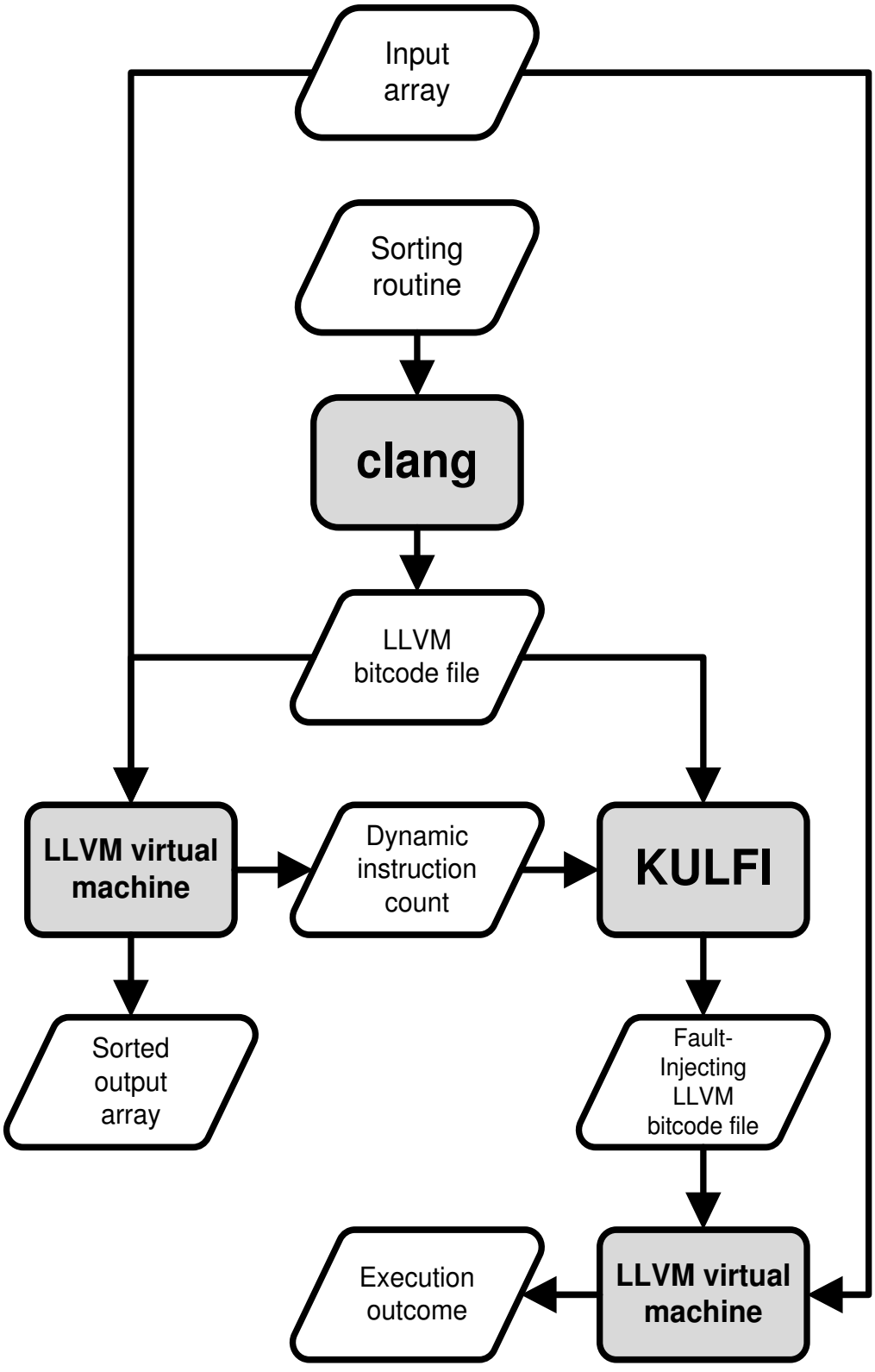

Figure 2.3: Fault injection strategy

these logs and present our empirical results next.

From Figures 2.4, 2.5, and 2.6, we observe that the values in log entries obtained after every fault injection campaign are strongly clustered, and there is a statistically significant distribution of the fractions for each outcome category. The larger shapes (e.g., triangles) in the middle of clusters are indicative of the greater number of instances of faults that are closer to the middle. More specifically, the fractions of every category of outcomes across the 200 fault injection experiments follow the 68-95-99.7 (or three-sigma) rule of normal distribution. Therefore, using our empirical data, we can draw statistically significant conclusions about the behavior of the analyzed sorting routines in a faulty environment.

Figure 2.7 details the comparison of the sorting algorithms based on the average number of executions in each category of fault outcomes. For example, we observe that BubbleSort, though an algorithm with higher time complexity than HeapSort, leads to more detectable faults. In fact, HeapSort is the least resilient to SDCs and results in either benign faults or SDCs in its fault injection campaign. QuickSort masks a majority of injected faults and therefore a high number of benign faults. It is worthwhile to note that the three algorithms that have the least number of detectable faults (MergeSort, QuickSort, and HeapSort) follow a recursive divide-and-conquer algorithm design paradigm. On the other hand, BubbleSort leads to more segmentation faults. We observe that approximately 85% of the executions of QuickSort and 90% of the executions of BubbleSort avoid SDCs. To sum up, QuickSort is the most resilient and available algorithm of the algorithms considered. The following summarizes the resilience-related observations, where the lower numbers are better (lower number of faults in those categories):

```
SDCs : Bubble < Quick < Radix < Merge < Heap
Seg. Faults : Heap < Quick < Merge < Radix < Bubble
```

In addition to logging execution outcomes, we also maintained a mapping of the dynamic instruction where the fault was injected to the outcome that was produced in that execution. We draw some interesting observations from this mapping. In BubbleSort, half of the faults injected into registers that are employed in computing the index of the array access in the expression `Array[i-1]` produce segmentation faults. In HeapSort, injecting faults into the instructions executed just before and immediately after the recursive calls causes

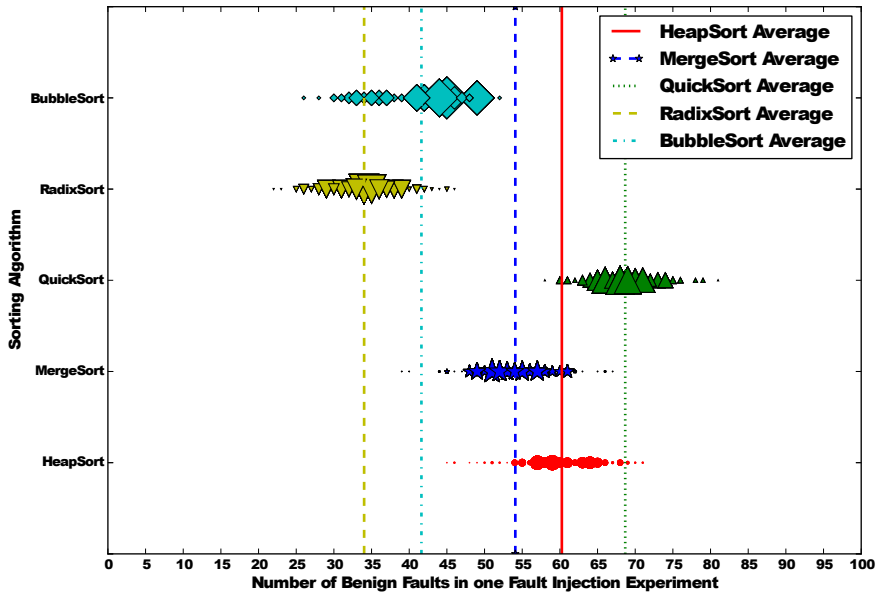

Figure 2.4: Benign faults

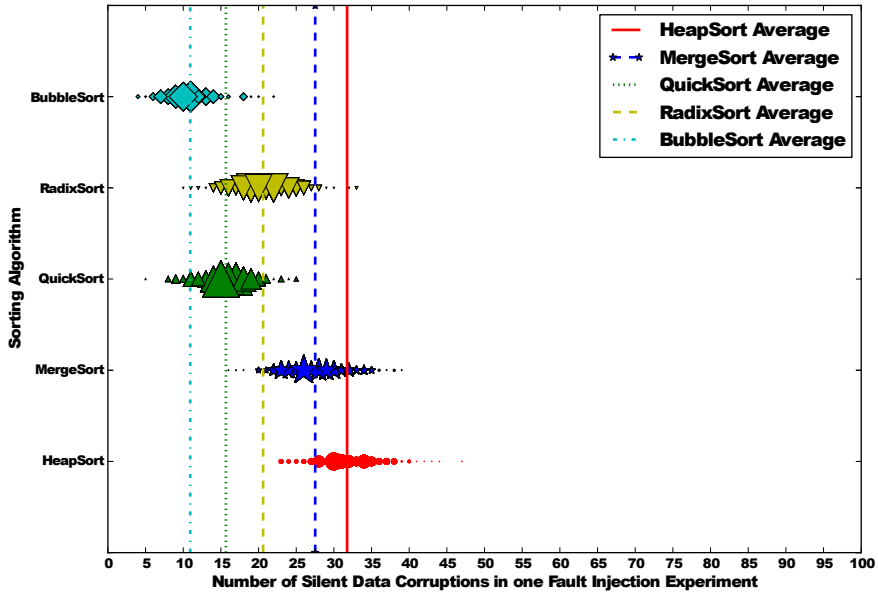

Figure 2.5: Silent data corruptions

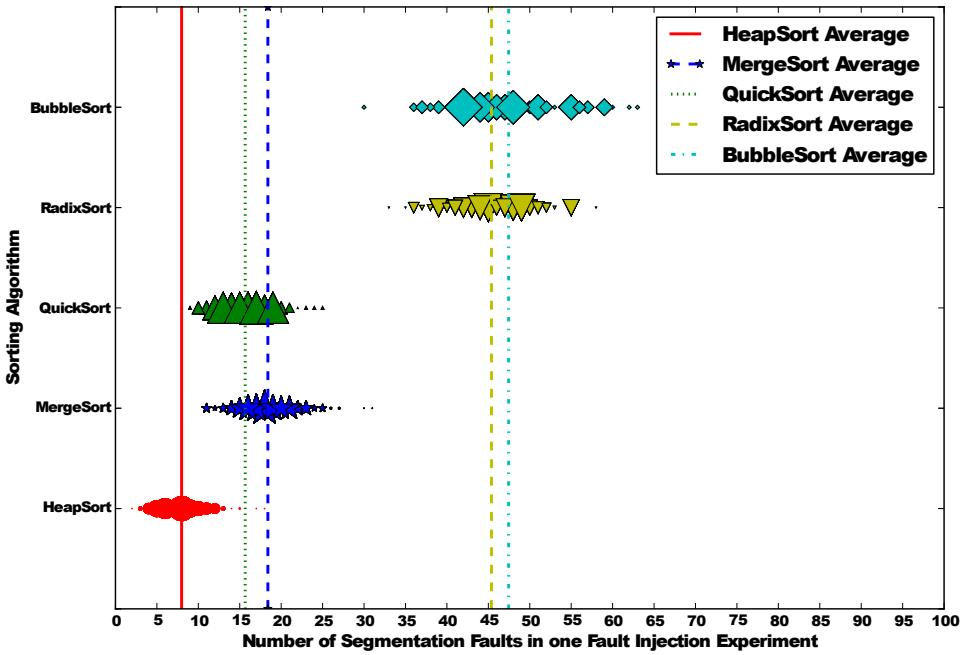

Figure 2.6: Segmentation faults

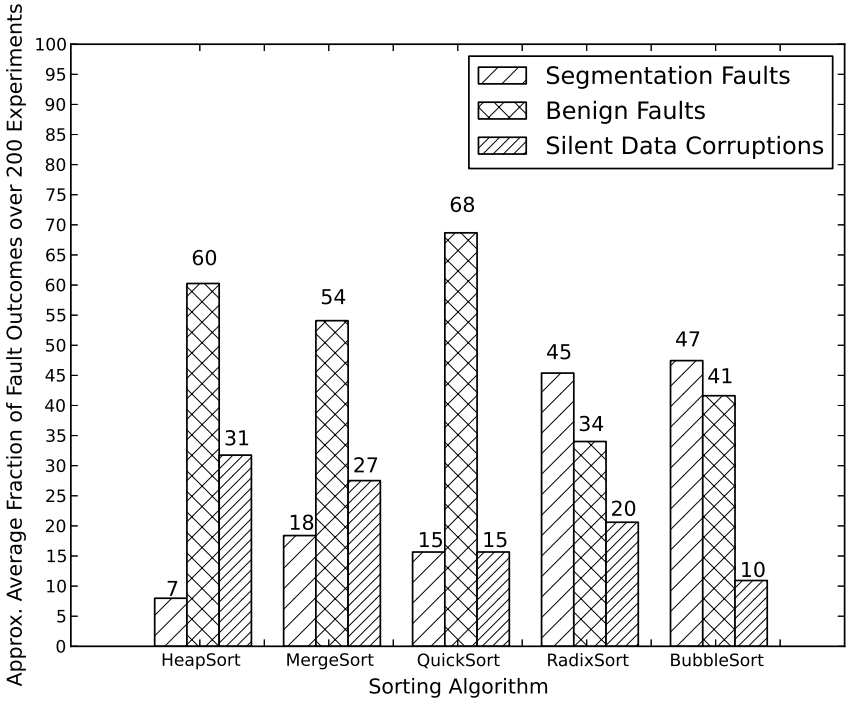

Figure 2.7: Summary of fault injection campaigns

a high percentage (close to 75%) of SDC faults. Such precise profiling of fault injection sites enables us to observe critical instructions, specific to a sorting algorithm, where error injection leads to SDCs. Note that one can potentially extend the notion of such critical regions of an algorithm to other algorithms that follow similar design patterns. Furthermore, targeted fault detection and recovery mechanisms can be employed around these critical regions to provide cheap and effective means for improving the resilience of programs to transient faults.

2.6 VULFI: A Fault Injector Handling Vector Instructions

Our initial effort in the form of KULFI tool gave us an excellent mechanism for evaluating the resilience of programs through a fault-injection driven approach which was demonstrated by evaluating the resiliency of five well-known sorting algorithms. However, with most of the commodity processors nowadays coming by default with vector pipelines in addition to scalar ones, there is an increasing focus to harness the raw potential of these vector pipelines for obvious performance gains. This has forced even mainstream compilers, such as GCC [2] and Clang [1], to aggressively employ vectorization during code generation phase. On top of that, there are specialized compilers such as ICC, ISPC, and OpenCL, which employ advanced vectorization techniques such as automatic loop parallelization which may require frequent enabling or disabling of a subset of the vector lanes. For example, the latest x86 architecture supports dedicated mask registers for enabling/disabling specific vector lanes. In these scenarios, a fault injector must track the status of mask register bits to count the active fault sites accurately. Clearly, KULFI was not designed to support vector architectures. Therefore, we developed a new fault injector named VULFI (**V**ectorization-aware **U**tah's **L**LVM-based **F**ault **I**njector). VULFI has been developed from the ground up with almost no code in common with the KULFI tool. While developing VULFI, we have applied our learning from the development effort of KULFI, and have also added several features which are not present in KULFI such as support for C++, ISPC, and OpenCL languages, and new options for fault site selection.[2]

[2]VULFI is publicly available at: `http://formalverification.cs.utah.edu/fmr/vulfi`

2.6.1 Terminology and Assumptions

We now define our terminology as well as some of our default assumptions.

Vector and scalar instructions: An LLVM IR instruction will be referred to as an "*instruction*." A *vector instruction* has at least one `vector` type operand.[3] A scalar instruction has no `vector` operand.

Vector and scalar registers: A *vector* register is an `Lvalue` register or a source operand of an instruction of `vector` type. A *scalar* register is an `Lvalue` register or source operand register of an instruction that has type `integer`, `floating point`, or `pointer`.

Vector length: The *length* of a vector register (V_l) is the number of scalar registers referred to within it.

`getelementptr`**:** At IR-level, the address of an element of an `aggregate` data-structure, such as an array, is calculated using `getelementptr` instruction.

Vector instruction – `extractelement`**:** It extracts a scalar element from a given location of a vector register.

Vector instruction – `insertelement`**:** It inserts a scalar element at a given location of a vector register.

Intrinsics: An intrinsic refers to a special function whose implementation is provided by the LLVM compiler infrastructure. All LLVM intrinsics start with a prefix `@llvm`.

Code generation, Architecture: We refer to IR-level code generated by the ISPC compiler with `-O3` optimization targeting x86.

2.6.2 Fault Site Selection Strategy

VULFI uses one of the following fault site selection heuristics to build an initial list of fault sites to be targeted for fault injections. Specifically, VULFI analyzes the forward slice of a fault site, to classify it into one of the following categories:

1. ***Pure-data*** **sites:** The forward slice of the fault site must not have any `getelementptr` (address calculation) or control-flow instructions.
2. ***Control*** **sites:** The forward slice must have at least one control-flow instruction.
3. ***Address*** **sites:** The forward slice must have at least one `getelementptr`.

For example, in Figure 2.8, a bit-flip occurring in the variable `i` may cause the loop

[3]The data types referred to in this work correspond to the type definitions provided in `http://llvm.org/docs/LangRef.html`

```
void foo(int a[], int n, int x)
{
        int s = x;
        for(int i=0;i<n;i++){
                a[i] = a[i] * s;
                s = s + i;
        }
}
```

Figure 2.8: An example C++ function `foo()`

execution to either end prematurely or run greater than `n` iterations. It may also become an out-of-bound index for the array `a[]`, thereby potentially causing an invalid memory reference. However, a bit-flip occurring in the variable `s` will never affect the loop control, neither will it cause an invalid memory reference. Therefore, the variable `i` is an example of both a *control* site and an *address* site whereas the variable `s` is an example of a *pure-data* site. Figure 2.9 more formally shows how these three fault site categories relate.

2.6.3 Instrumentation Work Flow

Figure 2.10 highlights the key building blocks in VULFI and Figure 2.11 further shows the instrumentation workflow of VULFI using an example of a vector register of length four with each of its elements considered a unique fault site. VULFI performs the following three key operations as part of the instrumentation process: (1) Iterates over each of the scalar elements in the cloned value of the vector register; (2) In each step, VULFI extracts an uninstrumented scalar element (represented by a white circle), performs instrumentation, and inserts the result (represented as a solid black circle) into the vector register; (3) Finally, VULFI replaces the original vector register with its new cloned and instrumented version, redirecting all the *users* of the original vector register. Figure 2.12 illustrates this on a masked vector *load* operation followed by a masked vector *store* operation. The vector *load* and *store* operations are done using the x86 intrinsics `@llvm.x86.avx.maskload.ps.256` and `@llvm.x86.avx.maskstore.ps.256`, respectively. VULFI maintains an inbuilt list of x86 intrinsics, which classifies whether any given intrinsic performs a *masked* vector operation. In the current example, this information is used to ascertain that both the intrinsics use the execution mask value of the `%floatmask.i`

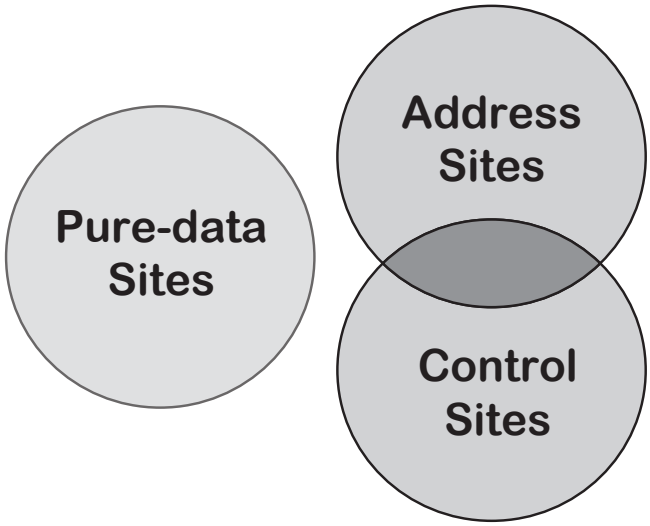

Figure 2.9: Relationship between different fault site categories

register to enable or disable *load* or *store* operations along the vector lanes. Figure 2.13 shows the instrumented version of the vector *load* and *store* operations. Each scalar element of the vector register `%0` (the chosen fault site for instrumentation) is extracted using `extractelement` instruction (locations `L1` and `L5`). The respective execution mask values of the scalar elements are extracted from the vector register `%floatmask.i` (locations `L2` and `L6`). An extracted element and its execution mask value is then passed on to the runtime fault injection API (`injectFaultFloatTy()` at location `L7` in the current example) to perform actual fault injection at runtime.

2.7 Evaluation of VULFI

2.7.1 Execution Strategy

The experiments described in the section were carried out on an Intel's Core^{TM}i7 4770 system running a 64-bit Ubuntu 12.04 operating system and with 16GB of main memory. The VULFI development is done with LLVM version 3.2, and the ISPC benchmark programs are compiled using ISPC compiler version 1.8.1. A *fault injection experiment* involves executing a benchmark program twice using a randomly selected program input chosen from a predefined set of inputs. During the first execution, no faults are injected, the execution output is recorded, and a dynamic fault site is chosen at random from a list of dynamic fault sites. The dynamic fault sites list is built by selecting either *pure-data* sites, *control* sites, or *address* sites. The second execution involves actual fault injection into the

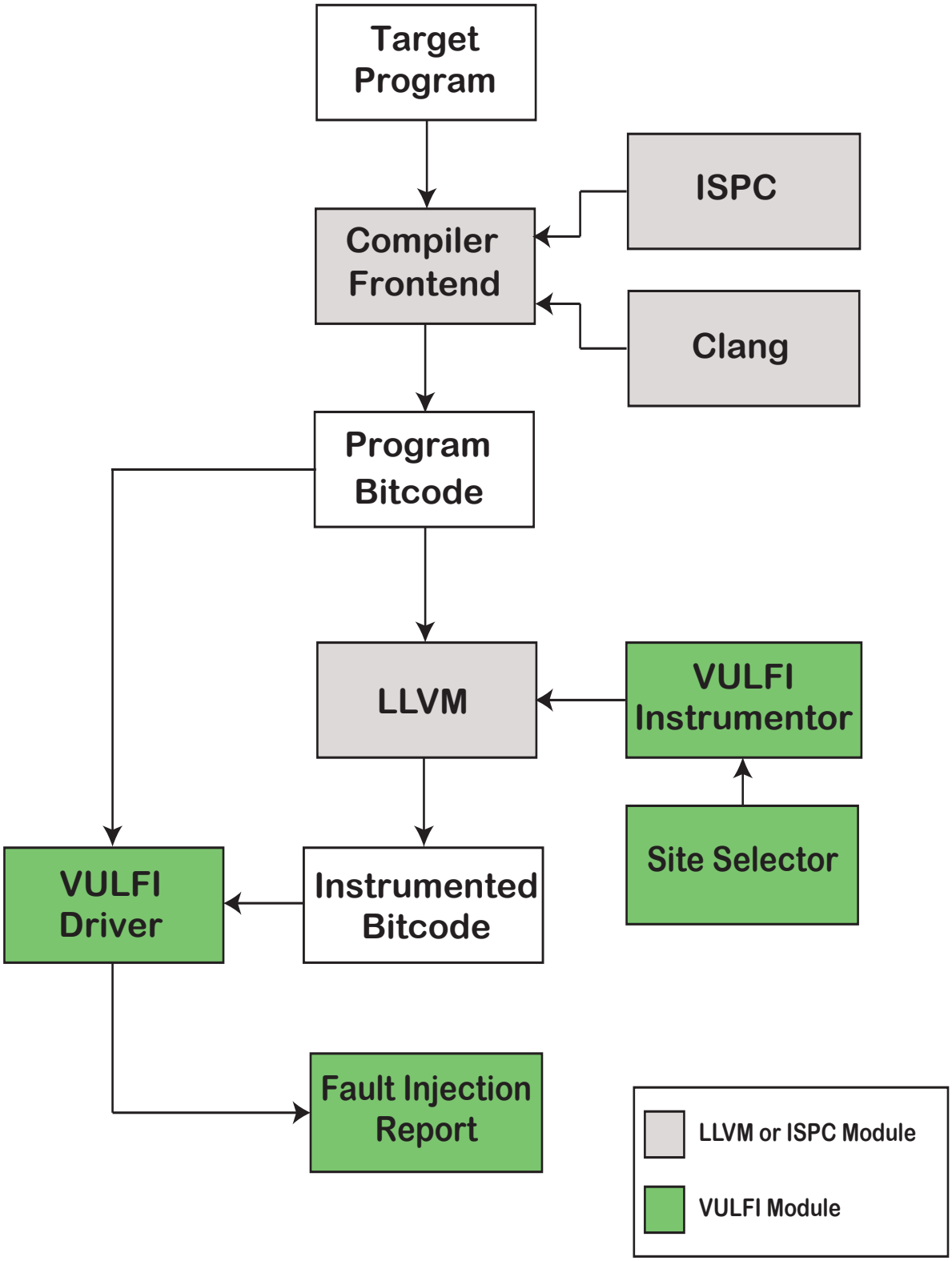

Figure 2.10: Building blocks of the VULFI framework

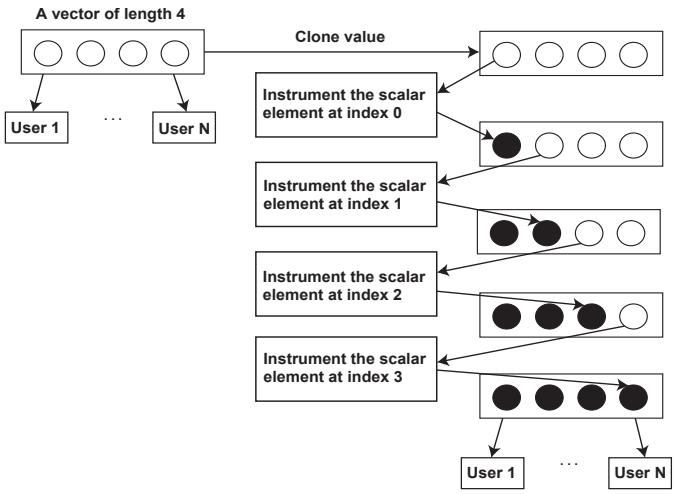

Figure 2.11: VULFI instrumentation workflow

```
L0: %0 = tail call <8 x float> @llvm.x86.avx.maskload.ps.256(
i8* %array_ld_addr, <8 x float> %floatmask.i)

L1: call void @llvm.x86.avx.maskstore.ps.256(i8* %array_str_addr,
<8 x float> %floatmask.i, <8 x float> %0)
```

Figure 2.12: Uninstrumented *masked* vector *load* and *store* instructions.

```
L0: %0 = tail call <8 x float> @llvm.x86.avx.maskload.ps.256(
i8* %array_ld_addr, <8 x float> %floatmask.i)

L1: %ext0 = extractelement <8 x float> %0, i32 0
L2: %extmask0 = extractelement <8 x float> %floatmask.i, i32 0
L3: %inj0 = call float @injectFaultFloatTy(float %ext0,
     float %extmask)
L4: %ins0 = insertelement <8 x float> %0, float %inj0, i32 0

...
...

L5: %ext7 = extractelement <8 x float> %ins96, i32 7
L6: %extmask7 = extractelement <8 x float> %floatmask.i, i32 7
L7: %inj7 = call float @injectFaultFloatTy(float %ext7,
     float %extmask7)
L8: %ins7 = insertelement <8 x float> %ins96, float %inj7, i32 7

L9: call void @llvm.x86.avx.maskstore.ps.256(i8* %array_str_addr,
<8 x float> %floatmask.i, <8 x float> %ins7)
```

Figure 2.13: Instrumented *masked* vector *load* and *store* instructions.

dynamic fault site chosen during the earlier run, using the error model described in §2.2.

2.7.2 Benchmarks

Table 2.2 lists nine benchmarks that we use for our fault injection experiments using VULFI. Benchmarks `fluidanimate` and `swaptions` are drawn from the PARVEC benchmark suite [21], and are vectorized implementations of their respective serial versions available to the PARSEC benchmark suite [12, 13]. Benchmarks `Blackscholes`, `Sorting`, `Stencils`, and `Ray tracing` are selected from the list of benchmarks available with the public release of the ISPC compiler. The remaining three benchmarks are our ISPC implementations of the respective C++ versions made available as part of the scientific computing library (SCL) by Burkardt [41]. For our fault injection studies, we target the fault sites corresponding to each vectorized function of these benchmark programs. The fault sites are selected using one of the heuristics explained in Section 2.6.2. The benchmarks are evaluated using `AVX` and `SSE4` as target `x86` vector instruction sets.

Figure 2.14 shows the mix of scalar and vector instructions for all nine benchmarks. A significant portion of instructions under *pure-data* and *control* fault site categories are vector instructions. Specifically, the number of vector instructions, averaged across all 9 benchmarks, stands at 67% and 43% for *pure-data* and *control* fault site categories. A seemingly low percentage of vector instructions under the *address* category should be taken with a grain of salt because at the IR-level, a scalar address is frequently cast into a vector address as and when required to be used by a vector instruction. *These details clearly highlight the importance of having a fault injector that can target vector instructions*—something we achieved by creating VULFI.

A *fault injection campaign* comprises 100 independent *fault injection experiments*. The SDC rate calculated for a *fault injection campaign* is considered a unique random sample. We run a sufficient number of *fault injection campaigns* until: (1) the sample distribution becomes normal or near-normal; and (2) for a target *confidence level* of 95%, the *margin of error* for the distribution falls within the range of $\pm 3\%$. We observe that for each of our benchmarks, running 20 *fault injection campaigns* for each of the fault site categories (*pure-data*, *control*, and *address*) is sufficient to achieve a 95% confidence level with a margin of error of $\pm 3\%$. More specifically, each row entry in the table refers to 60 fault injection campaigns, 20 each for *pure-data*, *control*, and *address* fault site categories (for both AVX and

Table 2.2: List of benchmarks used in the fault injection study

Benchmark	Program	Language	Test Input	Target	Average Dynamic Instruction Count (in millions)
Parvec	Fluidanimate	C++	sim_small sim_medium	AVX	170.8
				SSE	199.7
	Swaptions	C++	Swaptions: [16,64] Simulations: [100,200]	AVX	19.7
				SSE	16.0
ISPC	Blackscholes	ISPC	sim_small sim_medium sim_large	AVX	2.0
				SSE	1.9
	Sorting	ISPC	1D Array length: [1000,100000]	AVX	5.9
				SSE	5.4
	Stencil	ISPC	2D Array dimension: Min. – 16x16 Max. – 64x64	AVX	57.4
				SSE	69.3
	Ray tracing	ISPC	Camera input: Sponza, Teapot, Cornell	AVX	69.6
				SSE	68.8
SCL	Chebyshev	ISPC	Degree: [1,256]	AVX	1.8
				SSE	0.8
	Jacobi	ISPC	2D Array dimension: Min. – 32x32 Max. – 192x192	AVX	52.0
				SSE	44.5
	Conjugate Gradient	ISPC	2D Array dimension: Min. – 32x32 Max. – 256x256	AVX	45.6
				SSE	43.6

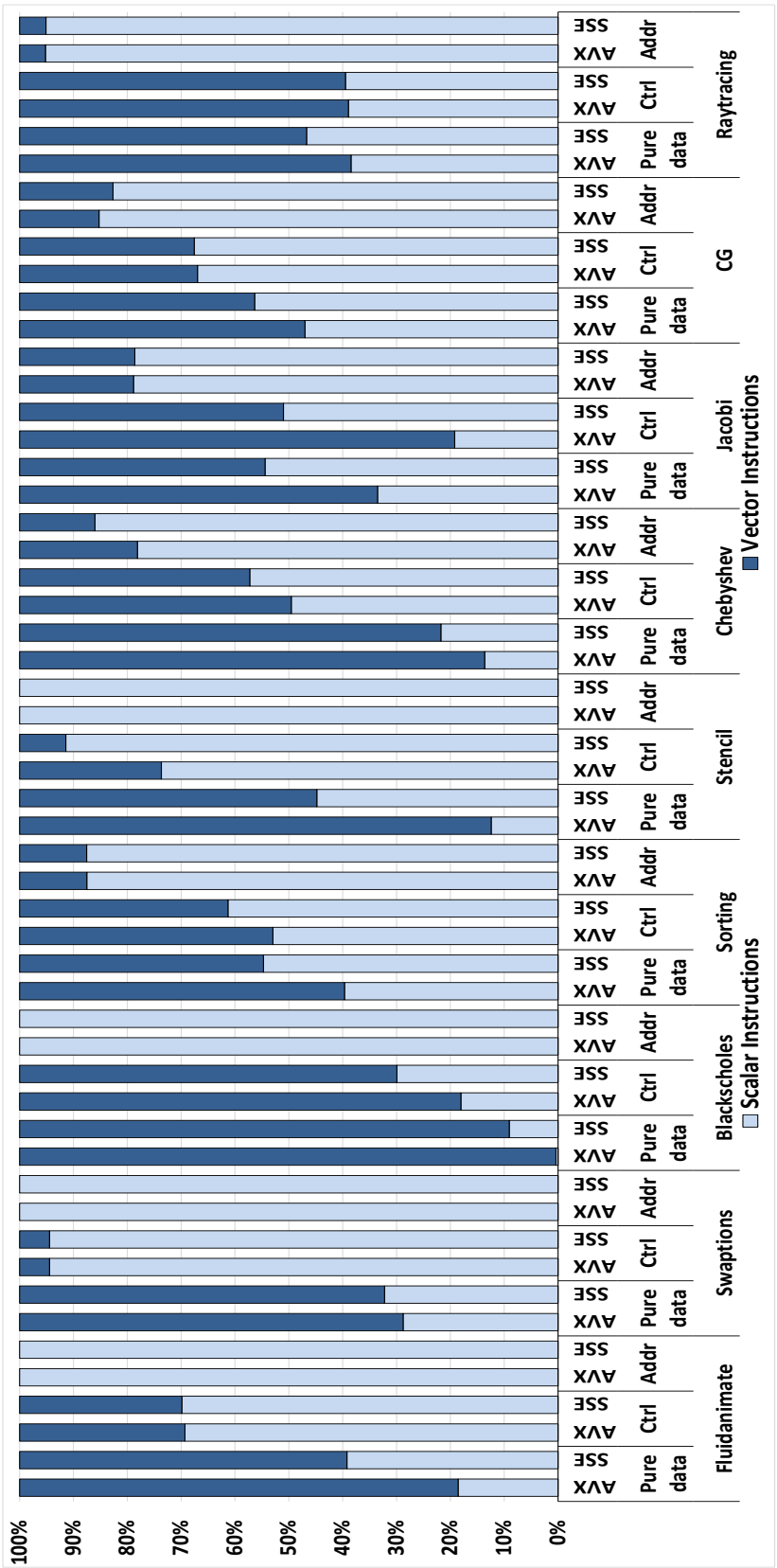

Figure 2.14: Composition of vector and scalar instructions in the benchmark programs

SSE4). This pushes the total number of fault injection experiments to $9 \times 2 \times 3 \times 2000 = 108,000$. The margin of error is calculated by applying the standard *t-value*-based formula where the sample size and the standard error of the sample distribution is known [111].

Figure 2.15 shows that among all benchmarks, `Stencil` and `Blackscholes` witness the highest rates of SDC. In contrast, `Swaptions` and `Conjugate Gradient` have the lowest SDC rates across all three fault site categories. Along expected lines (as these examples are array-intensive), the *address* fault site category results in the highest number of program-crashes. However, for benchmarks `Sorting`, `Stencil`, and `Chebyshev`, the *address* category also generates a significant number of SDCs. In fact, in the case of the `Chebyshev` benchmark, the SDC rate under the *address* category is highest among all three fault site categories. The result clearly establishes `Swaptions` and `Conjugate Gradient` benchmarks as the most resilient programs witnessing the lowest number of SDCs and crashes.

2.8 Related Work

Fault-tolerant computing has been very actively researched for decades, and forms the basis of many practical techniques in use, including redundant designs, voting schemes, and hardware-level error detection and correction schemes. There has also been an extensive study to identify and ameliorate fault-inducing mechanisms at the circuit level [89]. The manifestation of faults at the software level can be modeled by flips (changes) in bit-values of the computational state. In this work, we first compared various algorithms on the *k*-unsortedness metric. More specifically, the lower the number of misplaced data items, the more resilient the algorithm is deemed to be. In contrast, in our case study, we assume a more fine-grained error model that accommodates more fault categories, specifically, at the register and control-flow level. We also compare algorithms based on the number of silent data corruptions.

One of our main contributions is the development of a fault injector called KULFI, based on the LLVM compilation infrastructure [55, 58]. KULFI can inject transient faults into a chosen data register of a randomly chosen program instruction at runtime. Several previous studies have exploited fault injectors similar to KULFI [49, 99]. There are also efforts that directly inject faults into the hardware [61]. Hardware-based fault injection is less flexible [46] and not as programmable. Fault injectors can also be built by exploiting

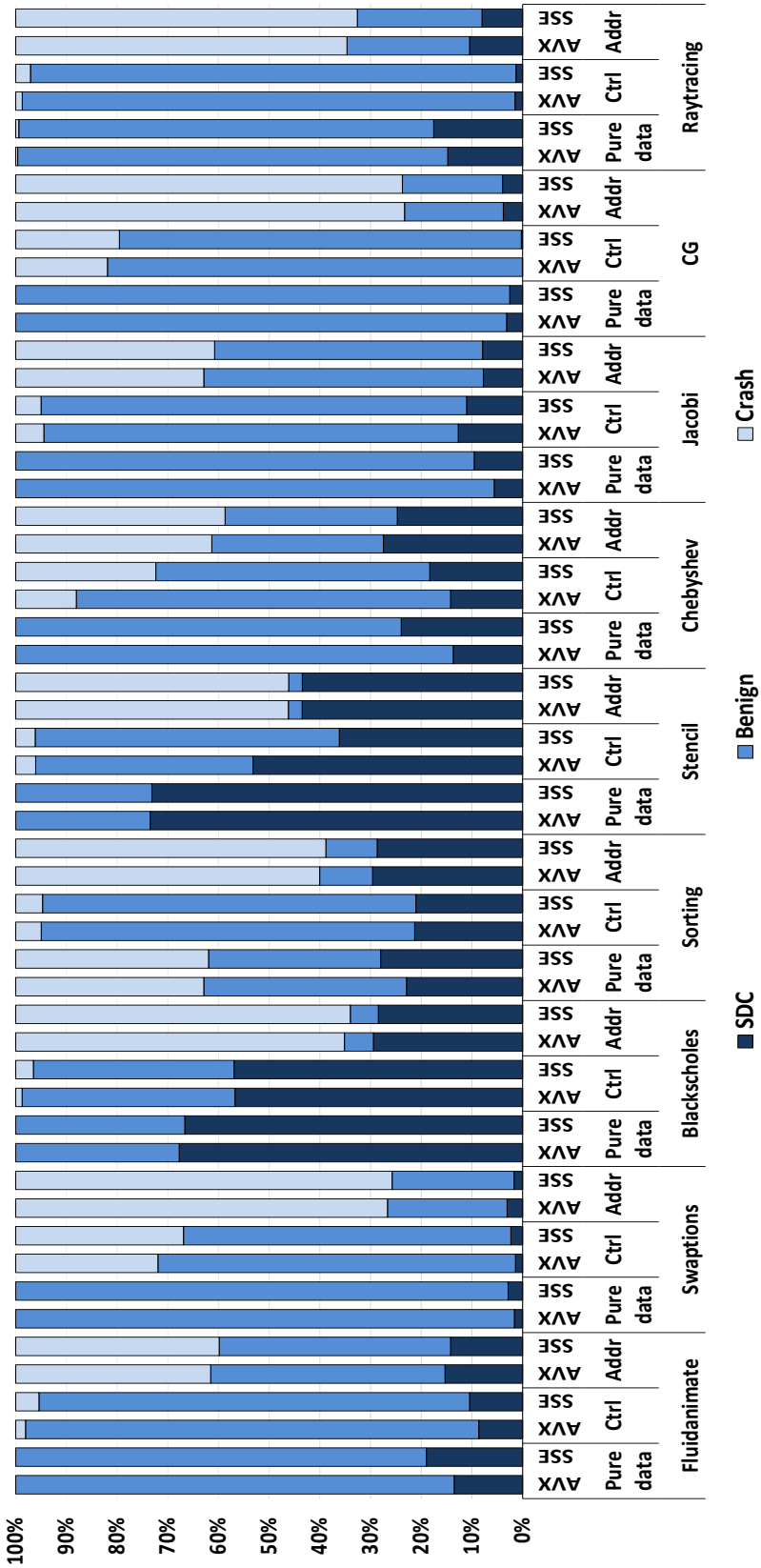

Figure 2.15: Result of fault injection experiments for the vector benchmark programs

OS-level facilities [49, 99]. Other software-level fault injectors include those based on PIN [31,48]. Specifically, the PDSFIS fault injector [48] uses Intel's PIN framework.

In contrast to the above works, KULFI uses the open-source LLVM compiler infrastructure, similarly to other recently reported fault injectors [31, 59]. The fault injector LLFI [59] is primarily geared towards injecting errors in soft-computing applications. LLFI and KULFI were developed concurrently, and currently, they share many similar features. However, when we started working on this project, KULFI was the only tool available to us that had all of the required features. For example, KULFI provides fine-grained error injection control (briefly discussed in the next section), which suits well our requirements for performing the experimental evaluations described in this chapter. The fault injector used in the Relax framework [31] also uses the LLVM compiler infrastructure. However, this fault injector is not publicly available. Furthermore, a previous informal study by a Relax user suggests that KULFI is easier to control and fine-tune, while also providing interesting command-line options not found in Relax [24].

Next, driven by the requirement of targeting vector architectures for fault injection, we developed a new tool called VULFI. A recent fault injection study, done by Hari et al., introduces an assembly-level fault injector SASSIFI built specifically for NVIDIA's CUDA architecture [44]. Another CUDA-centric fault injection study done by Fang et al. introduces a new fault injector GPU-Qin [37] which uses the CUDA debugger tool to insert breakpoints at program locations where fault injections are to be done. In contrast to these fault injectors, ULFI targets vector instructions at LLVM-level with the awareness for architecture-specific vector extensions such as Intel's AVX and SSE instruction sets. We believe that by targeting LLVM-level vector instructions, it allows VULFI to support any SPMD front-end compiler which uses LLVM as the backend for machine code generation. To summarize, to the best of our knowledge, we did not find any publicly available LLVM-based fault injectors targeting either scalar or vector architectures at the time when we started our resilience research.

2.9 Discussion

The fault injectors presented in this chapter are developed using LLVM compiler infrastructure and target LLVM IR-level virtual instructions for fault injections. The error models, implemented by these fault injectors, consider targeting a single instruction

out of the total number of dynamic virtual instructions observed at runtime for a given program. While this approach ensures an unbiased way of selecting a dynamic LLVM instruction, it has a limitation as an LLVM instruction may get translated into one or more architecture-specific instructions and different architecture-level instructions may have different execution cycle requirements. On the other hand, given that most of the LLVM instructions are architecture-agnostic, it allows these fault injectors to scale on various architectures supported by the LLVM infrastructure. Therefore, the fault injectors presented in this chapter provide a reasonable trade-off between architecture-level accuracy versus scalability.

Another point to consider is that the fault injection experiments done on sorting routines using KULFI target control and data fault sites. When a control fault site is targeted, the elements of an input array to be sorted remain corruption free but the output array may still finally be unsorted. In contrast, when a data site is targeted for fault injection, an element of the input array itself may get corrupted. In this scenario, the output array may appear sorted, but it has an element missing which is part of the original input array. In our experimental results, we categorize both these scenarios, where either the output array is unsorted, or the output array is sorted but not all of its elements belong to the original input array, as cases of SDC.

2.10 Conclusion

In this dissertation work, we first presented a unique, thorough case study of the resilience of several popular and widely used sorting algorithms. To be able to perform such an extensive resilience study, we first implemented a new, open source LLVM-level fault injector called KULFI. Faults injected by KULFI at the LLVM-level provide a reasonable fault model for actual hardware faults. Using KULFI, we performed an extensive empirical study that observed the behavior of sorting algorithms when faults are being injected. Based on the statistically significant results of this study, we drew informative conclusions about the resilience of these algorithms. Our empirical results aim to serve as guidance for software developers that have to take resilience into account when choosing an appropriate sorting routine for their task. Our next primary contribution in this dissertation work is a well-engineered fault injector named VULFI which is geared towards vector architectures. We demonstrated the effectiveness of VULFI by carrying out a fault

injection study on a set of 12 vector benchmarks drawn from Parvec, ISPC, and SCL benchmark suites. Our experimental results demonstrated that these vector programs when compiled produce a nontrivial fraction of vector instruction under three fault site categories – pure-data, address, and control. Finally, we observed a variation in SDC rates witnessed by these benchmarks. These results not only proved the robustness and utility of VULFI but also helped us in gaining insights about the resiliency of the vector programs.

CHAPTER 3

DETECTING CONTROL-FLOW DEVIATIONS INDUCED BY SOFT ERRORS

3.1 Introduction

Bit-flips affecting the runtime control-flow in a program may lead to the execution of an unintended program path, thereby potentially producing an incorrect result. Therefore, we present two novel approaches for detecting control-flow deviations caused in programs targeted to run on scalar and vector architectures, respectively. Specifically, the first approach is geared towards detecting control flow deviations in *scalar* architectures and builds on the basic tenets of *predicate-abstraction* [8, 28, 42], although we do apply abstractions that tend to reflect the degree of degradation of control-flows during program execution. To formally explain the control-flow deviations in a program caused by bit-flips, we found it natural to adopt a predicate-abstraction-based approach. Our overall goal is to observe and explain these bit-flips regarding their effect on abstract predicate state transitions. Therefore, we adopt the predicate-abstraction-based approach introduced by Ball [7]; while Ball used predicate-abstraction to define a novel program coverage metric, we use it to study faulty behaviors in a more manageable abstract state space.

In contrast, our second approach focuses on detecting control-flow deviations in vector loops. This approach exploits the code generation logic in the ISPC compiler [3,81] to mine invariants about specific control sites in vector loops. To the best of our knowledge, both the approaches, the first one using predicate-abstraction and the second one requiring invariant mining by analyzing code generation patterns in a vector compiler, to detect control-flow deviations, are novel research directions in resilience research.

3.2 Unique Research Contributions

Note that we have come up with these two approaches as part of two separate research efforts and therefore, they represent two disjoint ideas sharing a common thread of detecting control-flow deviations. Consequently, we have divided this chapter into two parts detailing the approaches as mentioned earlier.

Specifically, Part I described in Section 3.3 makes the following unique contributions:

- We present a novel abstraction method for programs executing in the presence of single-bit faults. We show how the abstraction explains the measured resilience results to some extent which is enough to serve as a way to rank one sorting algorithm above another.
- We propose a novel way to build error detectors using predicate-abstraction to detect the presence of soft errors causing single-bit faults affecting the runtime control-flow of a program leading to control-flow deviations. These detectors rely on tracking predicate transitions to detect *invalid* ones which only occur in the presence of faults affecting a control-flow. We demonstrate the effectiveness of these detectors by applying them to a set of well-known sorting algorithms.
- To make our proposed error detectors scalable, we further develop an error detection framework named FUSED (**F**ramework at **U**tah for **S**oft **E**rror **D**etection), developed using ROSE compiler infrastructure [84, 88], to automatically instrument a target program for extracting a set of valid predicate transitions which are then used as likely invariants to detect errors.
- We evaluate the scalability and the efficacy of the FUSED framework and its underlying technique, which involves using *predicate transitions* for error detection, by evaluating it on an open source LU factorization routine drawn from the SuperLU library [33, 57]. We report the detection rate and the average performance overhead witnessed by the error detectors which are automatically deployed using FUSED in the LU factorization routine.
- We also present a heuristic to identify highly sensitive code-blocks in the LU factorization routine of the SuperLU library, with a possible future application in optimizing the placement of the error detectors built by FUSED.

Section 3.4 presents our second approach of detecting control-flow deviations in vector

loops and makes the following unique contributions:

- We demonstrate extracting LLVM IR-level loop invariants for a `foreach` loop supported in the ISPC compiler to synthesize error detectors. Our findings highlight that the understanding of underlying code generation is central to discovering these invariants. We introduce our error models, set up our definitions, and provide an overview of our case studies.
- Also, we evaluate how well our detector types cover critical situations in practice. We also employ the LLVM IR-level loop invariants to build soft error detectors, reporting their efficacy and overhead using Intel's open source ISPC [3, 81] as the language and the compiler of choice.

3.3 Part I: Detecting Control-Flow Deviations Using Predicate-Abstraction

Given a set of predicates $\Phi = \{\phi_1, ..., \phi_k\}$ pertaining to program states, we can define abstract program states to be k-bit vectors representing the truth values of these predicates. As a running example, let us consider a three-variable program with three variables x, y, and z of type Int, and let us choose two predicates $x > y$ and $x + y = z$. Let $x = y = z = 0$ initially. One can define the *abstract* reachable state space of a program under predicate-abstraction as follows:

- The initial abstract state is obtained by assessing the predicates in the actual (concrete) initial state, thus obtaining a k-bit vector. In our example, the abstract state initial state is $\langle 0 > 0, 0 + 0 = 0 \rangle$, i.e., $\langle false, true \rangle$.
- For every concrete program transition, we update the abstract state reached suitably. Thus, a program statement `x++` executed in the initial state will transition to state $\langle 1 > 0, 1 + 0 = 0 \rangle$, i.e., $\langle true, false \rangle$.

Whenever a predicate is undefined in a state or one of the variables referred to in it is out of scope, the evaluation of the predicate is undefined, and recorded as `X`. It is clear that by evaluating each concrete state transition and applying the abstraction defined by the vector of predicates, we obtain a boolean state transition system. Typical predicates chosen for defining such a predicate-abstraction are the program conditionals and program invariants. For our studies in this dissertation work, we choose program conditionals. As our results show, this choice helps us understand control-flow deviations on non-faulty

executions.

3.3.1 Approach Overview

Table 3.1 shows the tracking of spurious *predicate transitions* to detect soft errors which cause a control-flow violation (case 2), and a data error (case 3). In the example, labels L0–L5 are the program locations, and PP0–PP4 are the program points. We choose the program conditionals $(x < 5)$ and $(y < 4)$ as the predicates. In general, these predicates govern the major control-flow steps in a program, and therefore are important to understanding a program's behaviors. The selected set of predicates defines our abstract states. We evaluate a vector of these predicates at chosen critical program locations, typically where one of the predicates governs control-flow, thereby effectively computing reachable abstract states at those locations. A predicate state concerning a program point PP is defined as PP:$\langle\phi_1\phi_2\rangle$, where $\langle\phi_1\phi_2\rangle$ is a bit-vector. In the bit-vector $\langle\phi_1\phi_2\rangle$, ϕ_1 and ϕ_2 represent the boolean values of the chosen predicates $(x < 5)$ and $(y < 4)$, respectively, at the program point PP. Using this definition, the concrete predicate state observed at the program point PP0 is PP0:FT as the predicates $(x < 5)$ and $(y < 4)$ evaluate to *false* and *true*, respectively. A predicate transition comprises a current state, a next state, and a transition from the current to the next state. For example, in case 1, the concrete predicate transition with respect to PP0 and PP3 is represented as PP0:FT $\rightarrow$ PP3:FT, where PP0:FT and PP3:FT are the concrete predicate states at the program points PP0 and PP3, respectively. In cases 2 and 3, a bit-flip (induced by a soft error) causes SDC in the program output, and spurious predicate transitions (both highlighted in red color) which are absent in the fault-free execution instance shown in case 1.

3.3.2 Another Example: Predicate Transition Diagram for BubbleSort

We present yet another example which leverages predicate-abstraction using BubbleSort. Figure 3.1 shows the source code of BubbleSort for which we generate a predicate transition diagram. We choose predicates $i > 0$, $j \leq i$, and $array[j-1] > array[j]$ from the BubbleSort routine, and analyze the effect of single-bit faults on the chosen predicates. In our example, we mark the program points in the BubbleSort routine starting with a label `PP0` and ending with a label `PP6`. The evaluation of a predicate at a program point depends on whether the predicate and the variables referenced by the predicate are in scope. For

Table 3.1: Motivating example

Case 1: Fault-free execution	**Case 2: Faulty execution**	**Case 3: Faulty execution**

`L0: int x = 5;`
`L1: int y = 3;`
`PP0:`
`L2: if (x< 5 && y < 4)`
`PP1:`
`L3: y = y + 2 ;`
`PP2:`
`L4: else`
`PP3:`
`L5: x - - ;`
`PP4:`
	`L0: int x = 5;`
	`//bit-flip at bit 0 in var x`
	`//new value of x = 4`
	`L1: int y = 3;`
	`PP0:`
	`L2: if (x< 5 && y < 4)`
	`PP1:`
	`L3: y = y + 2 ;`
	`PP2:`
	`L4: else`
	`PP3:`
	`L5: x - - ;`
	`PP4:`
		`L0: int x = 5;`
		`L1: int y = 3;`
		`//bit-flip at bit 2 in var y`
		`//new value of y = 7`
		`PP0:`
		`L2: if (x< 5 && y < 4)`
		`PP1:`
		`L3: y = y + 2 ;`
		`PP2:`
		`L4: else`
		`PP3:`
		`L5: x - - ;`
		`PP4:`

Program Output: x=4 , y=3	Program Output: x=4 , **y=5** (SDC)	Program Output: x=4 , **y=7** (SDC)
Predicate Transitions: PP0:FT $\rightarrow$ PP3:FT PP3:FT $\rightarrow$ PP4:TT	Predicate Transitions: **PP0:TT $\rightarrow$ PP1:TT** **PP1:TT $\rightarrow$ PP2:TF**	Predicate Transitions: **PP0:FF $\rightarrow$ PP3:FF** **PP3:FF $\rightarrow$ PP4:TF**

```
bubbleSort(array[], size){
PP0:  //state --> XXX
L0:   for (i = (size - 1); i > 0; i--){
PP1:     //state --> TXX
L1:      for (j = 1; j <= i; j++){
PP2:        //state --> TTX
L2:         if (array[j-1] > array[j]){
PP3:           //state --> TTT
L3:            swap(array[j-1],array[j])
PP4:           //state --> TTF
            } else {
PP5:           //state --> TTF
L4:            skip
            }
          }
PP:       //state --> TFX
       }
}
```

Figure 3.1: BubbleSort with encoded predicate states

example, at PP1 only the first predicate is in scope, and at PP0, none of the predicates are in scope (i.e., none of the variables appearing in any of the predicates are in scope). The truth values of predicates not in scope are recorded as X, which stands for "unknown." Having instrumented the BubbleSort routine with appropriate predicate evaluators, we run it with and without faults injected, observe the abstract state transitions made, and superimpose the transitions in a predicate transition diagram. Multiple such runs are performed, one for each permutation of a fixed-size input array, to accomplish a higher degree of coverage of all possible executions of BubbleSort. Our empirical approach, which generates the predicate transition diagrams dynamically, ensures that it indeed represents over-approximations of the reachable state space.

The outcome is a diagram shown in Figure 3.2. We use the following conventions in this figure:

- The solid (black) edges are *valid transitions* observed during both fault-free and faulty executions, under some input.
- The dotted (red) edges are *invalid transitions* that are never observed during fault-free runs; they are spurious state transitions caused by fault injection.

Note that in our current implementation, we carry out the described predicate evaluation concretely at runtime. The benefit of obtaining predicate transition diagrams with

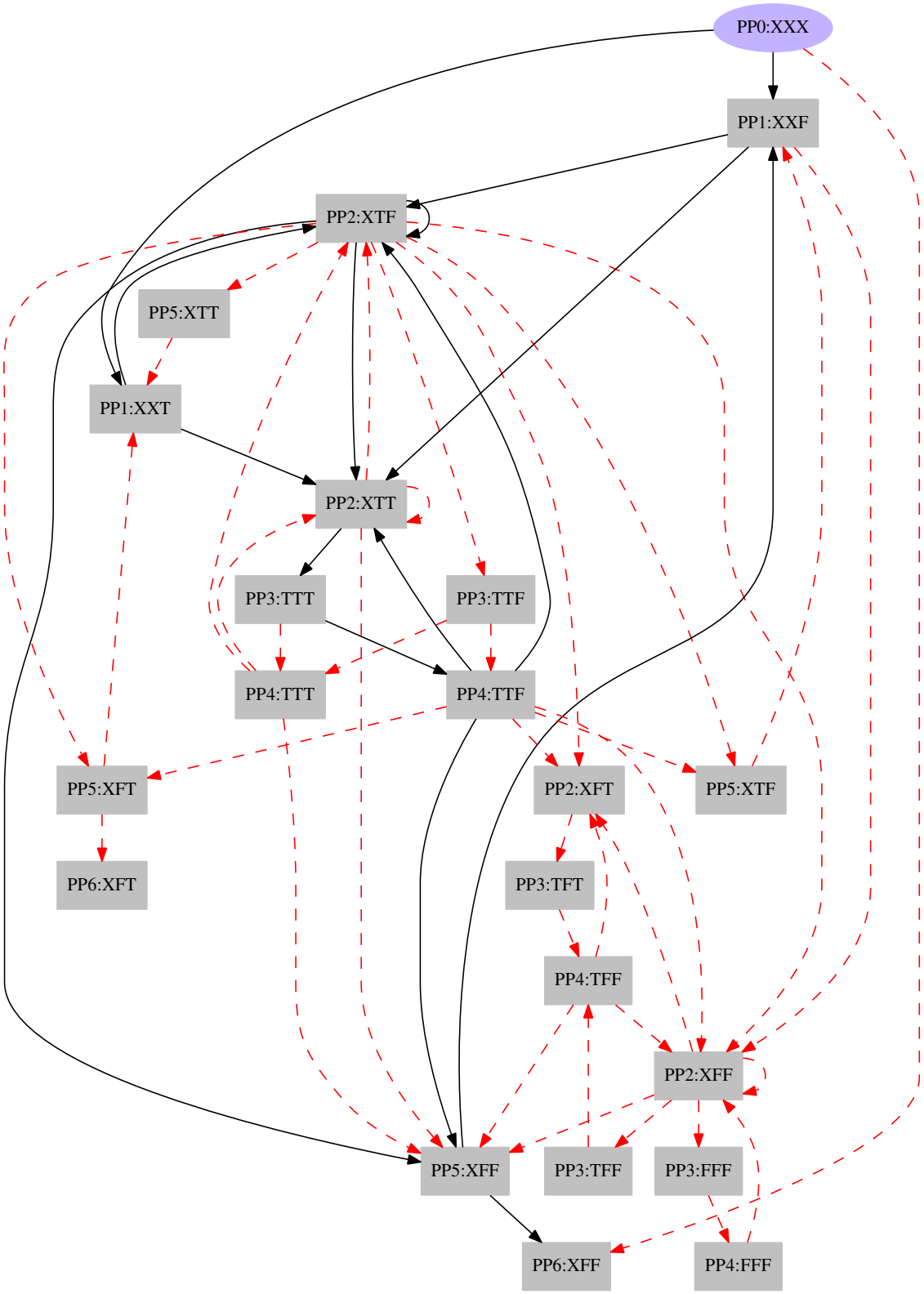

Figure 3.2: An abstract predicate transition diagram of BubbleSort

and without fault injection is that it provides an instantaneous visualization of the effect of faults on the overall program execution. Furthermore, Section 3.3.5 provides a preliminary assessment of our error detectors synthesized using the predicates employed in predicate transition diagrams. One of the continuing challenges in this line of research is to synthesize such error detectors that are extremely lightweight but still efficient enough in trapping the greatest number of faults.

3.3.3 Generating Predicate Transition Diagrams

The general approach of building predicate transition diagrams of the kind illustrated in Figure 3.2 is as follows:

- Given a collection of predicates and locations in a program, we instrument each of these locations with a call to a predicate evaluation function. This function takes predicates as inputs, evaluates them at the given program points, and returns a three-valued vector containing values true (`T`), false (`F`), and unknown (`X`) for each predicate. An unknown value is generated for a predicate if there is a variable in the predicate that is undefined at the program point.
- For a given program, let δ be its fault-free predicate transition relation and δ_F its predicate transition relation under faults. These transition relations are defined for a general program input.
- To combine these transition relations into a meaningful and informative predicate transition diagram, we create a solid (black) edge for every transition in δ and a dotted (red) edge for every transition in $\delta_F \setminus \delta$.

3.3.4 Experimental Results I

We now apply the procedure described earlier on all our sorting algorithms to generate their predicate transition diagrams. Then, we perform a preliminary assessment of the usefulness of the generated diagrams for estimating resilience of our algorithms. Figure 3.3 shows the abstract predicate transition diagram of QuickSort.[1] If we do a visual comparison of Figures 3.2 and 3.3, we can notice a higher degree of invalid transitions (dotted edges) in the BubbleSort diagram. This corresponds to the fact that the number

[1]Predicate transition diagrams for other sorting routines are provided at: `https://github.com/soar-lab/KULFI/wiki/Predicate-Transition-Diagrams-for-Various-Sorting-Algorithms`.

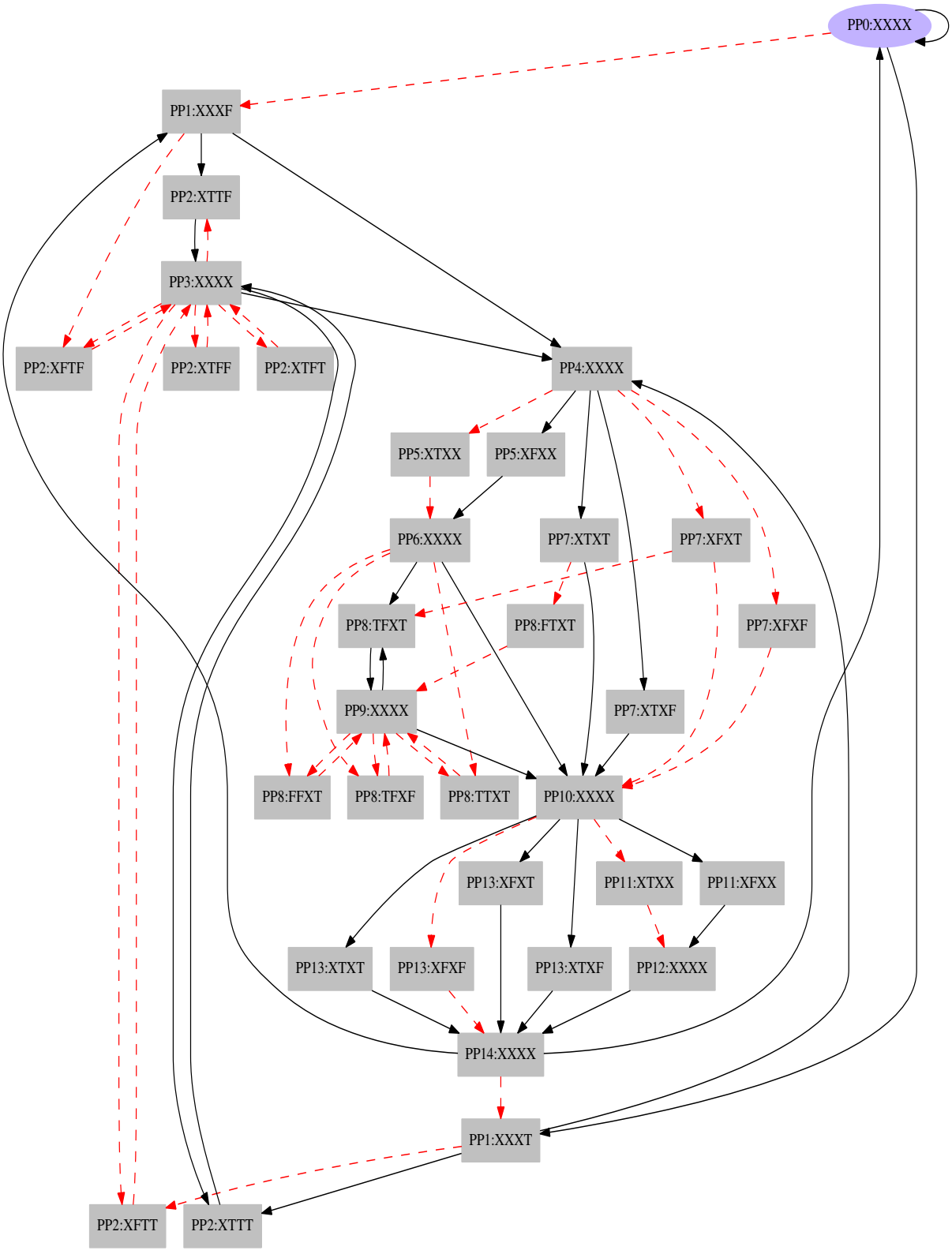

Figure 3.3: An abstract predicate transition diagram of QuickSort

of benign faults in BubbleSort is much lower than in QuickSort, which in turn leads us to compare the degree of valid transitions in our diagrams against the fraction of exhibited benign faults.

Table 3.2 provides statistics of the generated predicate transition diagrams on the number of valid and invalid transitions. We compared the percentage of valid transitions against the percentage of benign faults, and as it turns out, there appears to be a rough correlation between these numbers: a higher percentage of benign faults typically implies higher percentage of valid transitions. While this is a very preliminary, crude exploration that should be taken with a grain of salt, as an area of future work, we are planning to explore this and similar connections further. For example, we could refine our diagrams to include probabilities of particular transitions being taken, which would enable us to reason more precisely about how often particular invalid transitions are taken.

3.3.5 From Predicate Transition Diagrams to Error Detectors

In this section, we summarize our preliminary experiments that illustrate how error detectors can be synthesized based on insights gained from predicate transition diagrams. In a predicate transition diagram, the presence of invalid transitions may be leveraged to detect occurrences of single-bit faults. Hence, we devise a simple approach that uses generated predicate transition diagrams to detect faults occurring during execution of a sorting algorithm automatically. First, we compute a complete set of reachable abstract states for a fault-free sorting routine by running it on all possible inputs as described previously. Obviously, every subsequent run of the program under no faults must visit only states within this complete set. If not, we can surmise that the program execution has experienced a single-bit fault. These faults may result in a segmentation fault, turn out to be benign, or worse still, or result in an SDC.

To demonstrate the effectiveness of this approach, we use KULFI to test it on our

Table 3.2: Experimental statistics of predicate transition diagrams

Algorithm	Invalid Transitions	Valid Transitions	Total
BubbleSort	38 (71%)	16 (29%)	54 (100%)
RadixSort	64 (72%)	25 (28%)	89 (100%)
QuickSort	35 (53%)	32 (47%)	67 (100%)
MergeSort	67 (47%)	76 (53%)	143 (100%)
HeapSort	56 (66%)	29 (34%)	85 (100%)

sorting routines empirically. We run every sorting routine on an array of a given size where each element is initialized with a random value. During the execution, we inject exactly one fault using KULFI; also, we make sure that we only keep faults that cause SDCs and disregard others that we are not interested in. We also capture the set of reachable abstract states and check whether it is included in our initial complete set of abstract states. If not, we report an error detection. We perform these experiments on all five sorting routines with the size of input arrays fixed to 200; each routine is repeatedly executed until we obtain 1000 unique execution instances with each having SDC in its output.

Table 3.3 summarizes the error detection statistics. Note that the error detection statistics refer to the detection of only those faults which cause SDC in our experiments. The column "% Errors Detected" gives the percentage of faults (out of 1000 SDC-causing faults injected) that our approach successfully detected. The error detection percentage varies from 33.7% all the way to 100%, with an average of 77.6%, which clearly shows the promise of the approach. Clearly, sorting is an extreme example of a system where data (i.e., the items being sorted) directly affects control-flow. In general, the degree to which data affects control will determine the success of error detection based purely on control-flow tracking. Another point worth noting is that for a small-scale study such as ours, to build a reachable predicate state space, we execute a program on all possible inputs. In the future, and especially for larger experiments, we might use static analysis and/or sampling-based techniques for building the reachable state space. We will also incorporate lessons from previous work on control-flow tracking-based error detection cited earlier. Our hope is to bring the insight of predicate transition diagrams into this field.

Table 3.3: Error detection rates in various sorting algorithms

Algorithm	% Errors Detected
BubbleSort	66.0%
RadixSort	88.4%
QuickSort	100.0%
MergeSort	100.0%
HeapSort	33.7%
Average	77.6%

3.3.6 Automated Mining of Predicate Transitions

FUSED automates the task of source-level instrumentation for extracting likely invariants from a target program and inserting these invariants back into the target program for detecting errors. All instrumentations are carried out as source-level transformations using the ROSE compiler infrastructure. Specifically, these transformations are done by modifying the abstract syntax tree (AST) representation of the target program.

Figure 3.4 shows the two modes of operation for the FUSED framework: (i) *profiler* mode, and (ii) *detector* mode. The *profiler* mode is used to extract the likely program invariants in the form of a set of concrete predicate transitions. The *detector* mode is used for building and inserting error detectors in the target program.

3.3.7 Generating Likely Invariants

In the *profiler* mode, FUSED generates a separate *profiler* function for each function in the target program. The *profiler* function takes program points, and the concrete values of the boolean program conditionals of the target function, as inputs. The generated *profiler* function's definition is inserted into the target program. Function invocation code

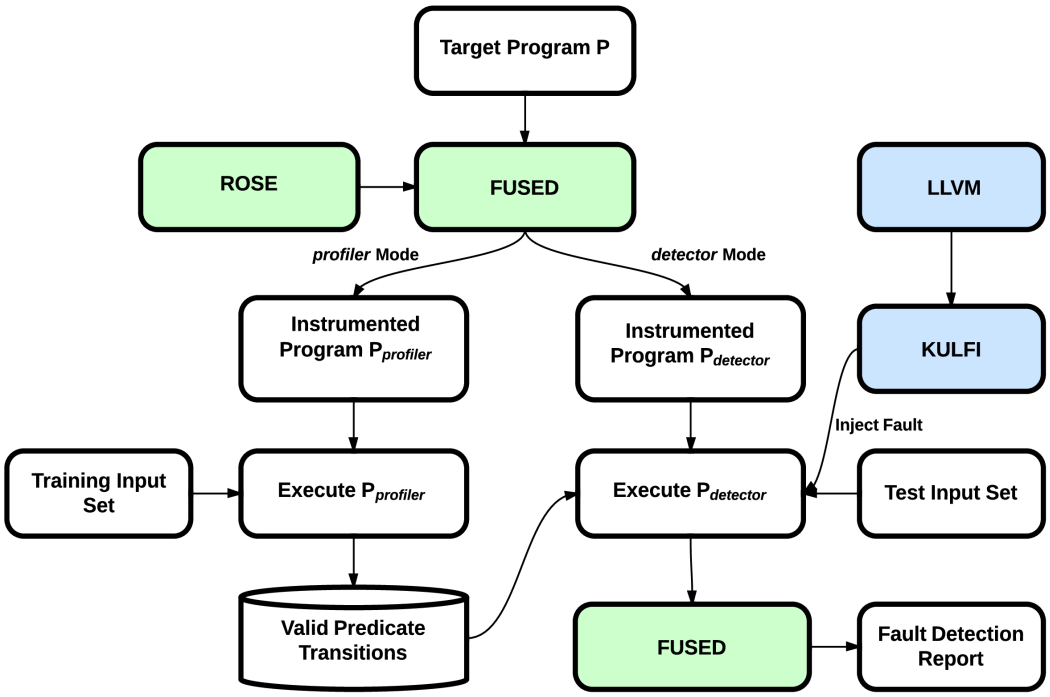

Figure 3.4: Workflow of FUSED framework

to call the *profiler* function is also inserted at user-selected locations in the target function. Fault-free executions of the instrumented program for a training input set produce a set of concrete predicate transitions. We call this set of concrete predicate transitions the *valid predicate transitions* which are used as likely invariants for runtime error detection.

Figure 3.5 formally presents the key steps involved in inserting *profilers* in a target program to extract likely invariants. Specifically, the function $ProfileTransitions()$ accepts a target program P and a *profiler* function $A_{profiler}$ as inputs. All the atomic boolean program conditionals involving non-pointers in the program P are chosen as a set of predicates Φ by calling the $SelectPredicates()$ interface of the FUSED framework. Program conditionals involving pointers when subjected to soft errors would most likely lead to segmentation faults and are therefore ignored. The profilers are inserted in P by invoking the $InsertProfilers()$ interface to generate output program $P_{profiler}$. The instrumented program $P_{profiler}$ is then executed for a preselected training input set I_{train}. Predicate transitions captured from each and every execution are aggregated into a set of *valid predicate transitions* δ_V which is used as the likely invariants by the error detectors.

3.3.8 Building Detectors

In the *detector* mode, FUSED generates a new *detector* function for each function in a target program. The *detector* function accepts the same set of input arguments as the *profiler* function, and is inserted in the target program at user-defined program points. Error detectors in the form of function invocations to the *detector* function are inserted in the target function. These detectors, when invoked at runtime, evaluate the currently observed concrete predicate transition, and check whether it is present in the list of *valid predicate transitions*. A concrete predicate transition missing from the list of *valid predicate*

```
1: ProfileTransitions(P,A_profiler) {
2:     Φ ← SelectPredicates(P)
3:     P_profiler ← InsertProfilers(P, A_profiler, Φ)
4:     for all I_i in I_train {
5:         S_pred_i ← Executed(P_profiler, i)
6:         δ_V_i ← GetPredicateTransitions(S_pred,i)
7:         δ_V ← δ_V ∪ δ_V_i
8:     }
9: }
```

Figure 3.5: FUSED algorithm for profiling valid predicate transitions

transitions signals the detection of an error, and is called an *invalid predicate transition*. Section 3.3.10 presents a heuristic to identify sensitive-code-blocks using these *invalid predicate transitions*.

Figure 3.6 shows the key steps involved in inserting error detectors in P where $A_{detector}$ is the actual *detector* function to be invoked at runtime to check for errors. The detectors are inserted into the target program P by invoking the $InsertDetectors()$ interface to generate the instrumented program $P_{detector}$. The program $P_{detector}$ is executed for a set of test input vectors I_{test}. During each execution iteration i, the set of observed predicate transitions δ_{O_i} is compared with the set δ_V. Any predicate transition in δ_{O_i} which is absent in δ_V is considered a *valid predicate transition* which is unprofiled during the *profiler* mode of execution, and is added to the set δ_V. The program $P_{detector}$ is then executed for the set of test input vectors I_{test} and faults are injected during the program executions. During each execution iteration i, the set of observed predicate transitions δ_{O_i} is compared with the set δ_V. The transitions in the set δ_I are called the *invalid predicate transitions* and used by the detectors to signal runtime error detection.

3.3.9 Addressing False Alarms

Our approach of using likely invariants (in the form of *valid predicate transitions*) suffers from a natural drawback of being prone to false alarms. To reduce false alarms for our experimentations, we extract the set of *invalid predicate transitions* observed during fault-free executions, and add them to the set of *valid predicate transitions*. In general, one good strategy for reducing false alarms is to use a larger sample size for the training input set to bring down the false alarm rate to an acceptable limit.

```
1:  DetectSoftErrors(P,A_detector) {
2:      Φ ← SelectPredicates(P)
3:      P_detector ← InsertDetectors(P, A_detector, Φ)
4:      for all I_i in I_test {
5:          S_pred_i ← Execute(P_detector, I_i)
6:          δ_O_i ← GetPredicateTransitions(S_pred_i)
7:          if δ_O_i contains invalid predicate transitions {
8:              ReportSoftErrorDetection()
9:          }
10:     }
11: }
```

Figure 3.6: FUSED algorithm for soft error detection

3.3.10 Identifying Vulnerable Code Regions

Figure 3.7 presents a heuristic to identify sensitive-code-blocks which under influence of soft errors are most likely to cause SDC in the program output. A set of *top invalid predicate transitions* δ^T is selected from the set of *invalid predicate transitions* δ_I based on their frequency of occurrence during the faulty program executions, by invoking the FUSED interface *GetTopInvalidTransitions()*. The elements in the set δ^T are further sorted in the order of their places of occurrence in the program $P_{detector}$ to obtain an ordered set δ^T_{sorted}. For each predicate transition, δ_i in the set δ^T_{sorted} which has an enclosing code block with one or more program statements, the corresponding program statements are added to the set $Stmt_i$. Each program statement in $Stmt_i$ is syntactically analyzed to see if they can influence the runtime concrete boolean values of the predicates in the set Φ. The program statements affecting the concrete boolean values of the predicates are added to the set $Stmt$. The set $Stmt$ is considered as the set of program statements in the program P which are most vulnerable to soft errors.

3.3.11 Experimental Results II

We evaluate the FUSED framework by assessing it on the sequential SuperLU scientific library. We choose the training and the test input sets for our experimentations from a variety of problem domains available in the University of Florida sparse matrix collection [30]. For our experimentations, we restrict the placement of the profilers and the detectors to the *dgstrf()* function in the SuperLU routine which is the core function responsible for

```
1:  IdentifyVulnerableCodeBlocks(δ_I) {
2:      δ^T ← GetTopInvalidTransitions(δ_I)
3:      δ^T_sorted ← SortByProgramLocation(δ^T)
4:      for all δ_i in δ^T_sorted {
5:          Stmt_i ← GetEnclosingCodeBlock(δ_i)
6:          for all stmt_j in Stmt_i {
7:              Var ← GetCommonVar(stmt_j, Φ)
8:              if Var is empty {
9:                  Stmt ← Stmt ∪ stmt_j
10:             }
11:         }
12:     }
13: }
```

Figure 3.7: FUSED heuristic for identifying vulnerable code blocks

performing LU factorization. In our experimentations, we consider placing the profilers and the detectors after every program statement of the *dgstrf()* function. Choosing this option helps us with our preliminary study of analyzing the influence of different program properties on the set of concrete predicate transitions observed in the faulty and fault-free execution traces.

Table 3.4 lists the set of input matrices used in our experiments and their classification based on the problem domain. In our experimentations, we carry out 20 fault injection campaigns for each of the input categories. Each of these fault injection campaigns comprises 1000 runs which sum up to 20,000 fault injection experiments for each of the five input categories to obtain a statistically significant result. We use KULFI, an instruction level fault injector for fault injections [54, 96].

Before starting the fault injection campaigns, the outputs from the fault-free executions of the SuperLU library with all the test inputs are recorded as *golden* program outputs. We also profile the set of *valid predicate transitions* by running the instrumented version of the SuperLU library with profilers inserted after every program statement of its *dgstrf()* function. The training input matrix for the *profiler* mode of execution is chosen at random from the respective input category.

During each run of a fault injection campaign, an instrumented version of the SuperLU library with detectors inserted after every program statement in its *dgstrf()* function is executed. The test input matrix for the *detector* mode of execution is chosen at random from the respective input category and is different from the one used during the *profiler* mode of execution. During the program execution, the detectors use the previously generated set of *valid predicate transitions* to detect soft errors injected using KULFI at runtime. Finally, FUSED compiles the error detection statistics at the end of all those runs in which an SDC

Table 3.4: Input classification

Problem Domain	Input Category	AvgDIC	Minimum Input Size
Optimization I	bp	262k	822 x 822
Optimization II	str	122k	363 x 363
Computational Fluid Dynamics	cavity	92k	317 x 317
2D/3D	fs_541	152k	541 x 541
Circuit Simulation	oscil_dcop	135k	430 x 430

is observed in the program output. During each run of the SuperLU library, we inject exactly one single-bit fault into a data register of a randomly chosen LLVM-level dynamic instruction using the instruction-level fault injector KULFI. The dynamic LLVM-level instruction is chosen with a probability of $\frac{1}{N}$, where N is the total number of LLVM-level dynamic instructions enumerated. This error model has been the model of choice for various resilience studies done in the recent past [31,96].

In Table 3.4, AvgDIC is the average dynamic instruction count. In general, the SuperLU library is resilient with a very low average rate of SDC in the execution output, ranging between 7.2 and 11.7 per 1000 fault injection experiments. The SDC rates, shown in Figure 3.8, are calculated by averaging the individual results obtained from the 20 fault injection campaigns per input category, and are statistically significant as they follow the three-sigma rule. The highest average SDC detection rate of 90.59% is observed for *cavity* input category. The average SDC detection rate across all the five input categories stands at 72.6% with an average execution overhead of 15.7%. An average false alarm rate of 35.6% is observed during the experimentations, which is addressed using the method described in Section 3.3.9.

Table 3.5 shows the list of sensitive-code-blocks obtained by applying the heuristic presented in Figure 3.7. Sensitive-code-blocks are the code regions in the program which, in the presence of soft errors, are most likely to cause SDC in the program output. Interestingly, the list of most sensitive program statements observed across all input categories is identical.

3.4 Part II: Error Detectors for Vector Loops

Vectorization is primarily supported in the form of: (i) language-specific vector extensions which are supported in modern compilers such as GCC [2] and Clang [1], and (ii) dedicated programming languages with inbuilt support for data-level parallelisms such as ISPC and OpenCL [73] that can enable vectorization to occur more predictably and under programmer control. Thus, languages such as ISPC and OpenCL, and their associated compilers, must be part of the research focus; we found no prior work targeting this angle. Also, there is an intriguing possibility that compiler writers of such languages have taken care to explain their code generator, and even provided some hints on the invariants being followed when specific situations are handled – for instance, partial vec-

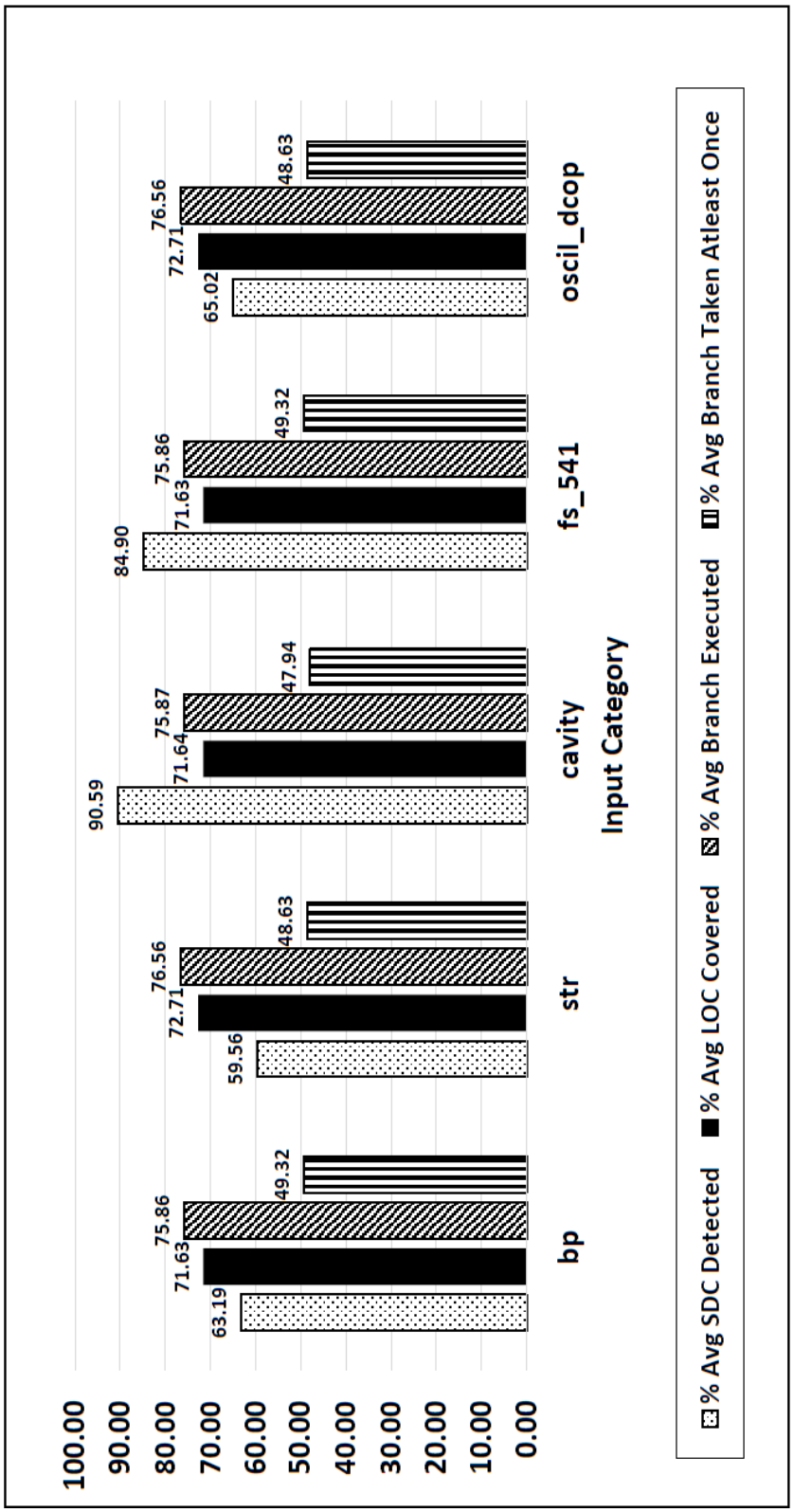

Figure 3.8: Error detection rates and coverage metrics

Table 3.5: Sensitive code blocks

Line#	Program Statement
280	kcol = relax_end[jcol];
306	k = xa_begin[icol];
330	k = jcol + 1;
349	k = (jj - jcol) * m;
384	jcol += panel_size;

torization supported by bit masking. We invested a significant amount of time studying ISPC's publicly available code generator (well-supported in this effort by the compiler team who answered our questions promptly), and found that *it offers another intriguing wrinkle regarding fault detector synthesis*: namely that these invariants could be turned into lightweight error detectors. This achieves two purposes at once:

1. One may be able to exploit specific patterns during code generation to tune and generate low overhead error detectors. While no single error detector type is sufficient to trap all types of faults (and our detectors are no exception), the attraction of error detectors that incur low overheads, are effective at trapping many faults, and (last but not least) *can be automatically generated and inserted* is potentially of huge interest in transferring resilience research into practice.
2. The dialog between the resilience research community and the compiler community may be a two-way street in that knowing the needs of the resilience community, compiler writers may be encouraged to document their compiler-backends more and provide features that support the generation of even more low overhead resilience solutions.

We demonstrate extracting IR-level loop invariants for a `foreach` loop supported in the ISPC compiler to synthesize error detectors. Our findings highlight that the understanding of underlying code generation is central to discovering these invariants. In Section 3.3.11, we evaluate how well our detector types cover important situations in practice. We also employ the IR-level loop invariants to build soft error detectors, reporting their efficacy and overhead, using Intel's open source ISPC as the language and the compiler of choice. As an initial step, we first describe two specific instances of how compilation methods can be exploited to generate error-checking invariants based on the code-generation logic of compilers.

3.4.1 Example 1: Loop Invariants in a `foreach` Loop Construct

An ISPC `foreach` loop accepts one or more *dimension variables*[2]of `integer` types with the iteration space of each *dimension variable* bounded by an interval $[start, end]$. A `foreach` loop uses its *dimension variables* as iterators to iterate over the loop body. To maximize lane utilization, for a given *dimension variable*, ISPC uniformly distributes first $\{n - (n\%V_l)\}$ loop iterations across V_l vector lanes, where $n = end - start$. The rest of the $n\%V_l$ loop iterations are handled separately.

Consider Figure 3.9 which presents the control-flow graph (CFG) of the *vector copy* function, `vcopy_ispc` of Figure 3.10. The `uniform` qualifier appearing in `vcopy_ispc` denotes that all vector lanes share the same address of arrays `a1` and `a2`, as well as the variable `n`. The `foreach_full_body` basic block executes $\{n - (n\%V_l)\}$ times with all V_l vector lanes performing parallel copy operations. The remaining $n\%V_l$ loop iterations are done in the basic block `partial_inner_all_only`. The values $\{n - (n\%V_l)\}$ and $n\%V_l$ are represented by the *definitions* `aligned_end` and `nextras`, respectively, in the entry basic block `allocas`. The *definition* `new_counter` is the loop iterator for the `foreach_full_body` basic block. Based on these facts, one can construct the following loop invariants for `foreach` constructs, as shown in Figure 3.11. Clearly, such invariants must always hold within, as well as upon exit from, `foreach` loop appearing in an ISPC program (to minimize overheads, we check them only upon exit).

3.4.2 Example 2: Protecting `uniform` Variables

In ISPC, a `uniform` variable is shared across all vector lanes. The compiler achieves this by storing a `uniform` value first into a scalar register and then broadcasting it to a vector register. In Figure 3.12 (typical result of compiling a code block containing a `uniform` variable), `uval` is a scalar register storing a `uniform` value. This value is first copied to the first element of the vector register `uval_broadcast_init`, and subsequently broadcast to all other locations using `shufflevector` instruction. The resultant value is stored in the vector register `uval_broadcast`. A bit-flip affecting any of the scalar elements of `uval_broadcast` can be detected by inserting a piece of checker code which ensures that all scalar elements hold the same value before every read from `uval_broadcast` (inexpen-

[2]In this work, we have considered `foreach` loops with only one *dimension variable*, but our findings are applicable to the cases with more than one *dimension variables*.

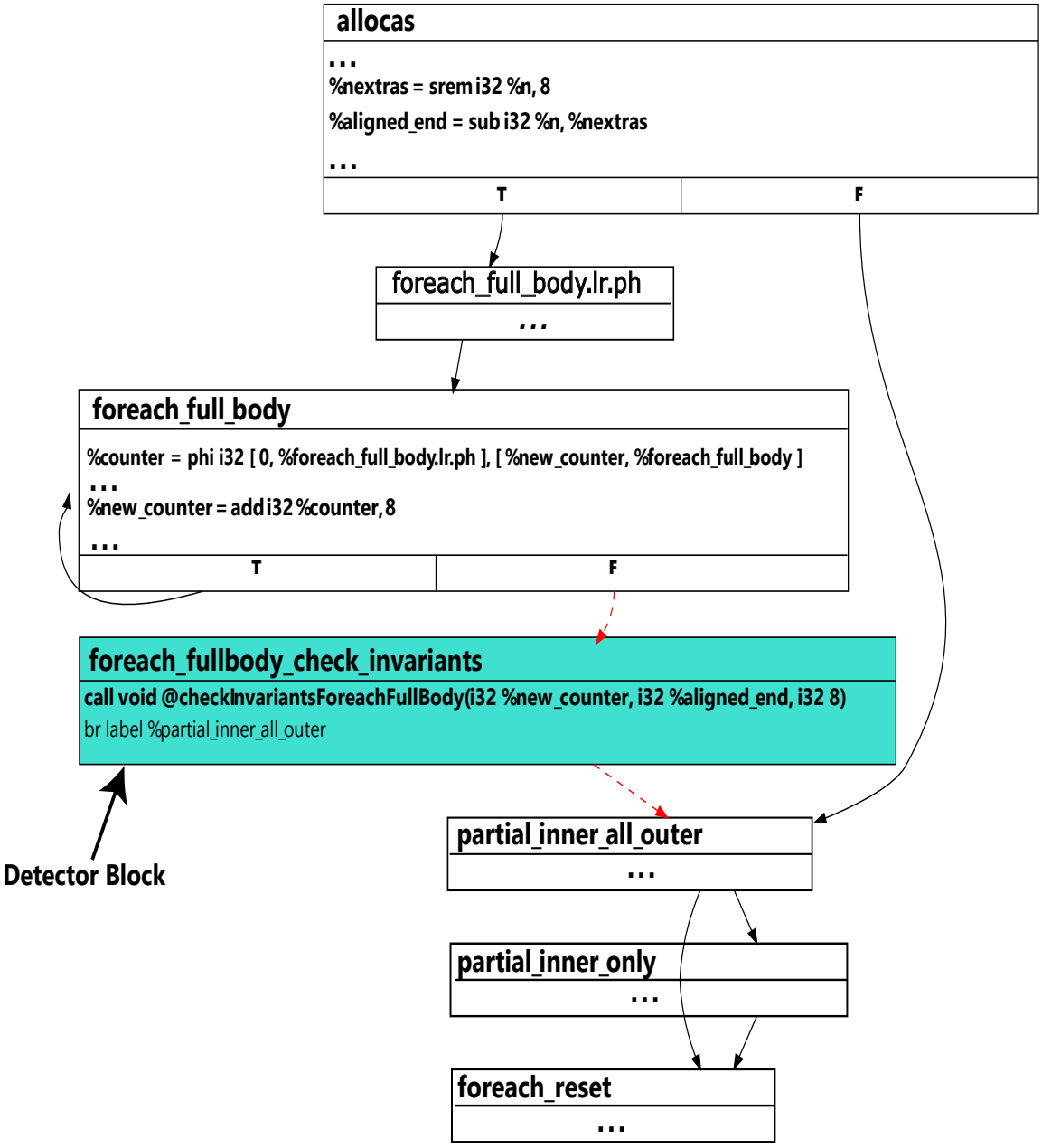

Figure 3.9: Control-flow graph of the `vcopy_ispc()` function

```
void vcopy_ispc(uniform int a1[], uniform int a2[], uniform int n){
      foreach(i = 0 ... n){
            a2[i]=a1[i];
      }
      return;
}
```

Figure 3.10: ISPC implementation of vector copy

```
Invariant 1: new_counter ≥ 0
Invariant 2: new_counter ≤ aligned_end
Invariant 3: (new_counter % V_l) == 0
```

Figure 3.11: Loop invariants for `foreach_full_body` basic block

```
%uval_broadcast_init = insertelement <8 x float> undef,
float %uval, i32 0

%uval_broadcast = shufflevector <8 x float> %uval_broadcast_init,
<8 x float> undef, <8 x i32> zeroinitializer
```

Figure 3.12: Broadcasting the value of the `uniform` variable `uval` to a vector register

sively achieved by XORing.) Such detectors can provide good, though not perfect, error detection coverage at very low cost. We have implemented a prototype LLVM transformation pass implementing the detector described in Section 3.4.1. Our prototype implementation automatically inserts a *detector* basic block for each occurrence of `foreach` loop in a program (Figure 3.11 highlights this block, namely `foreach_fullbody_check_invariants`). This block contains a *call* instruction which calls our runtime detector API that takes `new_counter`, `aligned_end`, and V_l as arguments. As noted earlier, we invoke the detector block only upon loop exit.

3.4.3 Error Detection Study

We evaluate the efficacy of the error detectors based on `foreach` loop invariants described in Section 3.4.1 on micro-benchmarks *vector copy*, *vector dot product*, and *vector sum*. Due to the (relatively) smaller size of these benchmarks, we follow a more comprehensive evaluation strategy by carrying out 2000 *fault injection experiments* for each of the micro-benchmarks under each of the fault site categories `pure-data`, `control`, and `address`. The detector's effectiveness is measured in terms of percentage of *fault injection experiments* that ends up in SDCs, together with the number that get flagged by our detectors.

The loop invariants in Figure 3.11 depend on the IR-level loop iterator `new_counter`. The value of this loop iterator is used to evaluate the loop exit condition, and also to calculate the addresses of the array elements referenced in the micro-benchmarks. Therefore, fault sites affecting the loop iterator will be categorized as either a *control* site or an *address* site or both, but can never be a *pure-data* site adhering to the relation shown in

Figure 2.9. Figure 3.13 confirms our hypothesis, showing that no SDCs are detected when *pure-data* sites are targeted for fault injection. In contrast, faults affecting *control* sites lead to the highest SDC rates, namely 96.5% for *vector sum*. In addition, 48.7% of the total *fault injection experiments* that end up in SDCs are also successfully detected.

Overall, the highest SDC detection rate is witnessed under *control* site category, with detectors approaching a detection rate of 57% for both *vector copy* and *dot product* micro-benchmarks. Faults affecting *address* sites report a relatively low SDC rate because a substantial number of fault injection experiments end up in program-crashes.

The overhead incurred by our detectors is measured by executing and comparing the runtimes of an instrumented program binary with and without the detector block inserted. Average overhead is calculated by averaging the overhead data from 2,000 individual runs for each micro-benchmark. A low average overhead of approximately 8% is witnessed across all three micro-benchmarks. We believe that with increasing operation counts inside the `foreach` loop body (compared to our very short loop bodies), the overhead introduced by these detector blocks will further get amortized. The examples presented earlier and the preliminary results discussed here have been quite encouraging, and it certainly opens up future avenues for exploiting compilation-aware detectors.

3.5 Related Work

There has been a significant amount of research on optimizing the placement of the error detectors [43, 75, 109, 112]. Research on using compiler-based techniques to detect hardware faults has also been reported [76, 113]. In a recent work [47], the focus is primarily on fault recovery (not on error detection) and custom annotations in the source language to convey the degree of resilience desired. Casas et al. [20] make a large-scale numerical application resilient by employing redundancy methods to guard pointers, and showing that some algorithms can recover from faults. Sahoo et al. [90] introduce an approach that employs likely program invariants for detecting hardware faults. SymPLFIED [78] is a formal framework that uses symbolic values to represent hardware faults, and performs a symbolic execution to simulate the propagation of such faults. It analyzes fault propagation patterns to optimize the placement of fault detectors.

Several techniques have been proposed to detect transient faults that cause control-flow variations. The work by Oh et al. [72] is based on assigning a unique signature to a

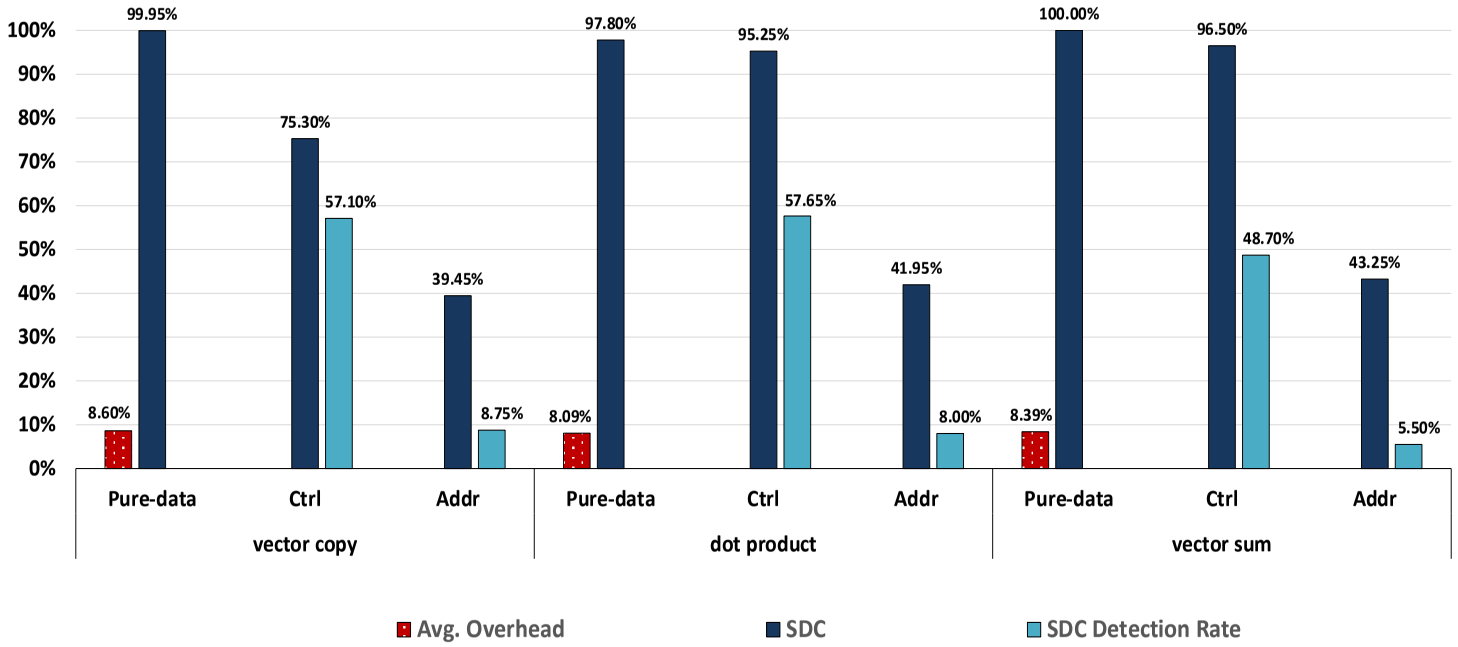

Figure 3.13: SDC detection rate on micro-benchmarks using the invariants-based detectors

basic block and tracking the signatures at runtime. Subsequently, Venkatasubramanian et al. [107] proposed a more refined and optimized solution. In contrast to these approaches, our technique is based on the use of predicate-abstraction [42].

Hardware faults evading hardware and micro-architectural level protections cause random bit-flips in the application computational states. These hardware faults introduced can either be permanent or transient in nature. Transient hardware faults (a.k.a. soft errors) typically persist for a short duration, and are often attributed to cosmic radiations and alpha particles [65,86]. Transient hardware faults occurring in the CPU state elements can cause application hang, crash, or silent data corruption (SDC) in the execution output, and are harder to detect as compared to permanent hardware faults. This has led to a recent upsurge towards developing low-cost application-level detectors as an alternative to hardware-level solutions.

There has been an effort to develop application-level detectors by monitoring software-level program properties and extracting likely invariants [90]. Another work involves similar approach for building error detectors in the form of program assertions by analyzing program properties and related rules on a set of dynamic execution traces [79]. Techniques are also proposed to build application-specific detectors by analyzing program properties where the majority of the SDC causing soft errors are visible [43]. Another approach builds architecture-level detectors by tracking history of the processor states on static instructions of a program. The history information is then used for detecting soft errors occurring in the processor elements [85]. Software-level detectors based on pure control-flow tracking have also been developed [72]. Algorithm-based error detection techniques like checksum-based approach in matrix multiplication [29], or techniques for localizing errors in the execution output of a linear solver, have also been introduced [101].

Unlike all previous approaches described here, both the detectors presented in this chapter pursue novel research directions. While our first approach is based on the idea of predicate-abstraction used by Ball to derive novel program coverage metrics [7], our second approach is based on our observation that during intermediate (e.g., LLVM-level) code generation, to handle full and partial vectorization, modern compilers exploit (and explicate in their code documentation) critical invariants.

3.6 Discussion

The predicate-abstraction-based detectors require constructing predicate state space where each state represents concrete values of a collection of boolean predicates at a given program point of a target program. A set of *valid* predicate transitions are then profiled by running the program on a set of training inputs and these *valid* predicate transitions are used later for error detection. Clearly, the efficacy of these error detectors are dependent on the completeness of the *valid* predicate transitions and the training inputs. In other words, any *valid* predicate transition not discovered during profiling phase would be incorrectly considered as *invalid* predicate transitions and therefore would lead to *false-positives*. Also, these detectors, in general, inherit scalability challenges from the predicate-abstraction technique. Specifically, the predicate state space grows rapidly with the number of predicates in a program. Therefore, targeting predicate-abstraction-based detectors in large programs may require employing additional predicate refinement strategies, which are not in the scope of this dissertation.

3.7 Conclusion

This work first introduced our novel predicate-abstraction-based approach for analyzing resilience of programs. In our approach, we dynamically generate abstract predicate transition systems for fault-free and faulty executions. Then, we superimpose these systems in a meaningful way to generate abstract predicate transition diagrams that visualize fault propagation at the higher, abstract level. Our abstraction seems to often explain at the high level the empirically measured resilience of algorithms. We then leverage our predicate-abstraction-based approach to build a simple error detector, and we show its effectiveness on our set of benchmarks. Next, to make our predicate transition-based detectors scalable, we implemented our technique as a compiler-level tool called FUSED to automate the profiling of predicate transitions and insertions of error detector in a target program. We demonstrated the effectiveness of FUSED by evaluating it on a real-world example of the SuperLU library. We also presented a preliminary version of a heuristic for identifying sensitive-code-blocks at the source-code level.

Our other key contribution came in the form of error detectors for detecting control-flow deviations occurring in vector loops generated by the ISPC compiler. Despite the flurry of research underway in system resilience, very few solutions (including our past

contributions) have been adopted and put into practice. The ”sticker shock” of suffering a flat-out $\sim$25%–2$\times$ overhead (typical figures for various error detectors and dual modular redundancy) can be unpalatable; a practitioner might prefer going back to an older lithography, suffering fewer errors (and overhead). The burden of manually inserting detectors into the source code can also hinder adoption. Finally, the non-availability of detector types (and even the means for conducting studies) targeting vector instruction sets and SPMD languages further hinders adoption. While we cannot say that the detection overheads are still within the ballpark of unquestioned acceptance, the detectors prove to be surprisingly lightweight, can be automatically generated and inserted during compilation, and may, in the grand scheme of things, provide the right kind and level of the solution.

CHAPTER 4

EXPLORING THE DESIGN SPACE OF ERROR DETECTORS IN STENCIL COMPUTATIONS

4.1 Introduction

Chapter 3 highlighted that several high-performance computing (HPC) applications involve straight-line code sections with very high arithmetic intensities. For such classes of applications, control-flow detectors would provide a very limited coverage, and therefore they must be complemented with other types of detectors capable of providing adequate coverage. Stencil computations represent one such class of applications which are often used in the discretized solutions of partial differential equations (PDEs) representing many important and complex physical simulations such as heat diffusion, computational fluid dynamics, electromagnetism, seismic wave propagation, etc.

Therefore, we undertake an exploratory work focusing on stencil computations mainly to assess the complexities involved in building efficient error detectors for such a class of applications that involve data-intensive computations. We present a feasibility study by first presenting a novel technique for synthesizing error detectors for stencil computations by learning the computational patterns using machine learning, and evaluating its effectiveness through a fault-injection-driven study. Our technique involves choosing a subset of points in a given stencil to learn an approximate kernel which is a computationally *less expensive* approximation of the original stencil kernel. We run the approximate kernel alongside the original stencil kernel and compare their results to infer whether an error has occurred. Fundamentally, our approach is based on the principal of dual-modulo redundancy (DMR) where we employ redundant computations to detect errors. However, the key novelty of our approach is that we propose a machine-learning-based approach to derive a *less expensive* approximate kernel to be used for redundant computations. We demonstrate our technique and the feasibility study using a 25-point reverse-time-

migration (RTM) stencil [9]. Our specific contributions are:

- A novel approach that uses regression techniques to compute efficient approximations of a computational kernel used in a stencil computation.
- A systematic way to reduce the size of the training data used for generating regression models.
- Using a cross-validation-driven approach to estimate sample size, and to select features for building efficient error detectors.
- A C++ shared library named SORREL that helps evaluate/extend our technique.

4.2 A Case Study Using 25-point RTM Stencil

We choose a 25-point (RTM) stencil as implemented by McCool et al. [63] as the target stencil kernel for our case study. This example RTM stencil kernel, shown in Equation 4.1, uses a finite-difference approximation of the PDE described in Equation 4.2, and is second-order accurate in time and eighth-order accurate in space. Here, P is a three-dimensional array and $P_n(x,y,z)$ denotes the value of the pressure wave at coordinates (x,y,z) at time n, and v is the velocity of pressure wave which is constant for a given medium.

$$\begin{aligned} P_{n+1}(x,y,z) = 2 * P_n(x,y,z) - P_{n-1}(x,y,z) + v^2 \Big\{ C_0 \\ * P_n(x,y,z) + \sum_{k=1}^{k=4} \{ C_k * (P_n(x+k,y,z) \\ + P_n(x-k,y,z) + P_n(x,y+k,z) \\ + P_n(x,y-k,z) + P_n(x,y,z+k) \\ + P_n(x,y,z-k)) \} \Big\} \end{aligned} \tag{4.1}$$

$$\frac{\partial^2 P}{\partial n^2} = v^2 \left(\frac{\partial^2 P}{\partial x^2} + \frac{\partial^2 P}{\partial y^2} + \frac{\partial^2 P}{\partial z^2} \right) \tag{4.2}$$

As shown in Figure 4.1, to calculate $P_{n+1}(x,y,z)$, the RTM stencil kernel uses the values of the 25-point RTM stencil from the previous time-step. Our goal is to train a regression model that predicts the output $P_{n+1}(x,y,z)$ of a stencil evaluation given one or more of its inputs. The choice of which inputs to use for model training affects its accuracy and cost. Table 4.1 lists the choices we evaluate in this study. The output and the selected inputs, for a given stencil evaluation, together form a feature vector. We train a regression model on a feature vector by collecting the respective dynamic execution data over multiple executions using various training inputs. The resulting regression model computes an

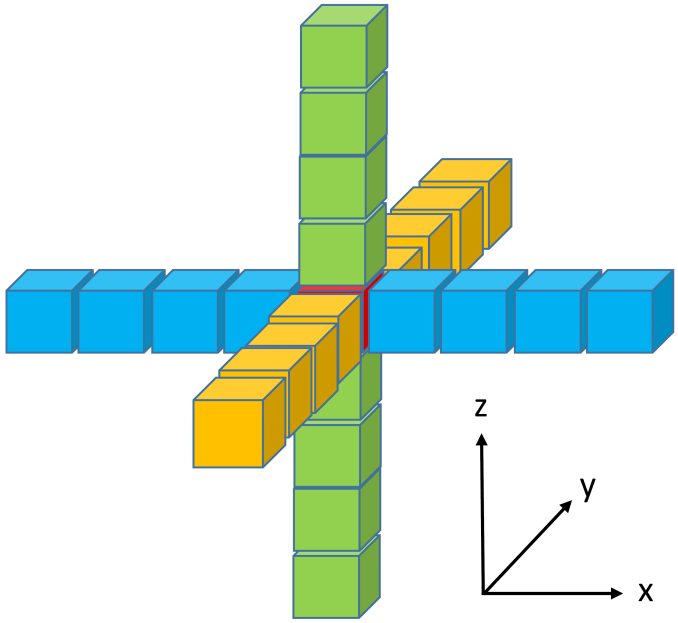

Figure 4.1: A 25-point RTM stencil

Table 4.1: Feature vectors based on 25-Point RTM stencil

Feature Vector	Feature List	Comments
f_1	$P_n(x,y,z)$	
f_2	$P_{n-1}(x,y,z)$	
f_3	$P_n(x,y,z)$,$P_{n-1}(x,y,z)$	
f_4	$P_n(x,y,z)$, $P_n(x+1,y,z)$,$P_n(x-1,y,z)$, $P_n(x,y+1,z)$, $P_n(x,y-1,z)$, $P_n(x,y,z+1)$,$P_n(x,y,z-1)$	
f_5	$P_n(x,y,z)$, $P_n(x+s_1,y,z)$,$P_n(x-s_1,y,z)$, $P_n(x,y+s_1,z)$,$P_n(x,y-s_1,z)$, $P_n(x,y,z+s_1)$,$P_n(x,y,z-s_1)$	$1 \leq s_1 \leq 2$
f_6	$P_n(x,y,z)$, $P_n(x+s_2,y,z)$,$P_n(x-s_2,y,z)$, $P_n(x,y+s_2,z)$,$P_n(x,y-s_2,z)$, $P_n(x,y,z+s_2)$,$P_n(x,y,z-s_2)$	$1 \leq s_2 \leq 3$
f_7	$P_n(x,y,z)$, $P_n(x+s_3,y,z)$,$P_n(x-s_3,y,z)$, $P_n(x,y+s_3,z)$,$P_n(x,y-s_3,z)$, $P_n(x,y,z+s_3)$,$P_n(x,y,z-s_3)$	$1 \leq s_3 \leq 4$

approximation of the RTM stencil kernel, and is used to verify whether the output of the original kernel is likely to be correct.

When selecting the features on which to train the regression model, our goal is to use as few inputs of the RTM stencil as possible while accurately approximating its output. Feature vectors f_1, f_4, f_5, f_6 include different subsets of the stencil's inputs, while vectors f_2 and f_3 include the stencil's output in the preceding time-step. In contrast, vector f_7 uses all 25 points in the RTM stencil and serves as an upper bound on the accuracy of a regression model. Our hypothesis, which is evaluated in the following sections, is that including more input data will produce a model that is more accurate but more expensive.

As a first step in training a regression model, we collect the data, which we use for training the regression model, by running the RTM application on randomly generated inputs. We produce the inputs by randomly varying the parameters such as array size, the number of time-steps, the value selected to initialize $P_n(x, y, z)$ at time $n = 0$, and the point of origin for the pressure wave denoted by $P_n(x, y, z)$ at time $n = 0$. During an execution of the stencil program, each point in time and space produces a unique observation. During the training phase, these observations are used to generate training data.

4.2.1 Sampling Technique

Given a large number of observations generated in the training phase, it is crucial that we perform sampling to reduce the training data size. As such, we employ stratified sampling [70] to collect a representative subset of all the collected data that captures the breadth of the full dataset but is tractable to train on. To sample, we divide the time-stepped computational data into a finite number of (preferably equal sized) strata. We then perform a simple random sampling of the computational data belonging to each stratum having a sample size expressed in Equation 4.3. Here, S_m is the sample size for a stratum with index m, and P_m is the corresponding population size. The population size P_m is the size of the complete computational data which belongs to the stratum at index m. The constant value k is the fraction of the total population size that is sampled.

$$S_m = k * P_m \tag{4.3}$$

4.2.2 Sample Size Estimation

To determine the number of samples necessary to compute an accurate model, we use a cross-validation-driven approach [74] that is illustrated in Figure 4.2. A target program is executed under a set of randomly-generated training inputs. Using an initial guess value for k, the raw computational data is sampled to produce an optimized training dataset. We perform n-fold cross-validation [53] on the training dataset by training LibSVM [22] on the training subsets and evaluating it on the test subsets to quantify accuracy. We iterate this process with the larger values of k until the respective cross-validation accuracy stabilizes. The smallest possible value of k that provides reasonable cross-validation accuracy is chosen for generating the regression model.

4.2.3 Regression Analysis

To generate approximate functions, we perform linear regression [5] using each feature vector listed in Table 4.1. Specifically, we use a linear regression kernel *epsilon*-SVR [27, 104, 105] implemented in LibSVM software [22]. The generated regression models are then used by SORREL to compute their corresponding approximate functions. In this

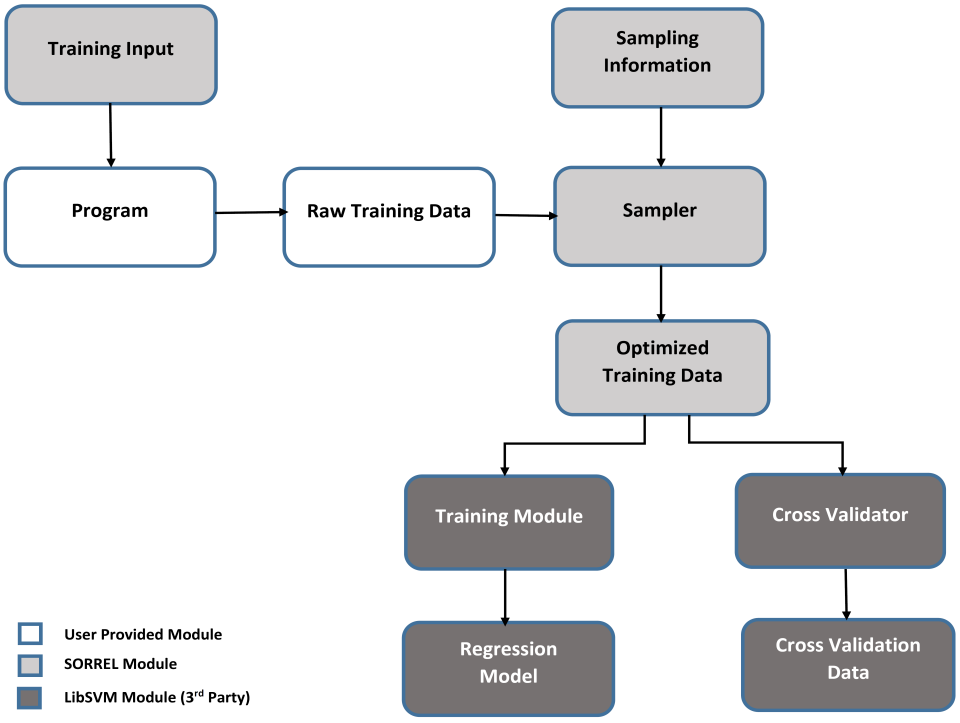

Figure 4.2: Sampler workflow during training phase

dissertation, we presented a novel approach using linear regression to efficiently approximate stencil kernels, validating it on the RTM stencil. We showed how the cost of training the model could be reduced via cross-validation-driven stratified sampling to reduce the training set size systematically. We also showed how to find a *near-optimal* threshold value(τ_{opt}) for the detectors using ROC curve analysis and presented the error detection rate and the overhead data for the detectors. A high error detection rate reported by our detectors demonstrates the effectiveness of our approach.

In Equation 4.4, $\mathcal{K}^n(x,y,z)$ represents a stencil kernel that computes $P_n(x,y,z)$, and $\mathcal{A}^n_{f_r}(x,y,z)$ is an approximation for $\mathcal{K}^n(x,y,z)$, generated through regression analysis using a feature vector f_r. As shown in Equation 4.4, $\mathcal{D}^n_{f_r}(x,y,z)$ is a boolean detector function. It returns *true* if the absolute difference between the observed and the predicted value for a given stencil computation, represented as $|\mathcal{K}^n(x,y,z) - \mathcal{A}^n_{f_r}(x,y,z)|$, exceeds a threshold value τ. Table 4.2 quantifies the additional number of operations performed by the detectors as compared to the *native* version of the RTM stencil kernel shown in Equation 4.1. The detectors D_{f_1} through D_{f_7} are synthesized using feature vectors f_1 through f_7, respectively, as listed in Table 4.1. Operation count mentioned for the detectors is in addition to the number of operations required by the *native computation*. As shown later in Section 4.2.7, the higher cost of detectors D_{f_4} through D_{f_7} relates directly to the larger numbers of RTM stencil points they use as input.

$$\mathcal{D}^n_{f_r}(x,y,z) = \begin{cases} \textit{true}, & \text{if } |\mathcal{K}^n(x,y,z) - \mathcal{A}^n_{f_r}(x,y,z)| > \tau. \\ \textit{false}, & \text{otherwise.} \end{cases} \tag{4.4}$$

Table 4.2: Comparison of operation counts

Operation Type	**Operation Count**							
	Native Computation	D_{f_1}	D_{f_2}	D_{f_3}	D_{f_4}	D_{f_5}	D_{f_6}	D_{f_7}
$\{*\}$	19	1	1	2	7	13	19	25
$\{+\}$	25	0	0	1	6	12	18	24
$\{-\}$	1	1	1	1	1	1	1	1
$\{>\}$	0	1	1	1	1	1	1	1
$\{!=\}$	0	1	1	1	1	1	1	1
Total	**45**	**4**	**4**	**6**	**16**	**28**	**40**	**52**

4.2.4 Threshold Estimation and Detector Accuracy

We now explain the method to determine an *optimal* or *near-optimal* value of the threshold (hereafter denoted as τ_{opt}) which yields a high error detection rate (*true-positives*) with few *false-positives*. To quantify the impact of τ on these two metrics, we plot a receiver operator characteristic (ROC) curve [16] for the detectors synthesized using feature vectors listed in Table 4.1. This curve shows the *true-positive* and *false-positive* rates achievable by each detector across a range of values for τ. For a given number of observations, the *true-positive rate* R_{tp} is calculated using the number of observations with *true-positives* (N_{tp}) and *false-negatives* (N_{fn}) as shown in Equation 4.5. The *false-positive* rate R_{fp} is calculated using the number of observations with *false-positives* (N_{fp}) and *true-negatives* (N_{tn}) as shown in Equation 4.6.

$$R_{tp} = \frac{N_{tp}}{N_{tp} + N_{fn}} \tag{4.5}$$

$$R_{fp} = \frac{N_{fp}}{N_{fp} + N_{tn}} \tag{4.6}$$

In the current context, an observation represents an instance of a program execution during which a soft error may or may not occur. If a soft error is witnessed during an observation and the soft error is successfully flagged by a detector, then the observation is regarded as a *true-positive* instance. However, if the detector fails to catch the soft error, then the observation is a *false-negative* instance. Conversely, during an error-free observation, if a detector falsely reports the detection of a soft error, then we treat this observation as a *false-positive* instance. If the detector does not flag any error during an error-free observation, then we call it a *true-negative* instance.

A ROC curve is obtained by plotting R_{fp} and R_{tp} against each other using x and y axes, respectively, across a range values for τ. We select the point on the ROC curve that gives a high value for R_{tp} and a low value for R_{fp}. Note that the acceptable values for R_{tp} and R_{fp} depend on the magnitude of errors a given application can tolerate, with some applications being inherently resilient to errors of low magnitude [101, 103]. We generate our error detectors in two phases. During the initial training phase, we generate

the approximate stencil function. During the test phase, we select τ_{opt} using ROC curve analysis and evaluate the detector's effectiveness.

4.2.5 Training Phase

We generate 1000 unique program inputs for the RTM program using SORREL. We use 1% of these inputs in the training phase, and the rest are used during the testing phase. Each of these training inputs leads to a huge amount of training data, underlining the need for sampling as explained earlier. The next step is to quantify the distribution of errors made by each model relative to the real values computed by the RTM stencil. To this end, we used n-fold cross-validation, where the set of observations (sampled using stratified sampling as explained in Section 4.2.2) is divided into n disjoint subsets. For each subset i, we train a model using the remaining $n-1$ subsets and compute the error of the model using subset i as the test data. Finally, we compute the overall error (hereafter referred as e_t) by applying the mean-squared-error (MSE) metric on the individual model errors obtained using each of the n subsets.

Figures 4.3 and 4.4 show the distribution of instances of e_t obtained using 10 different observation samples, when using either $n = 2$ or $n = 10$ and $k = 4e-5$. The data shows that the instances of e_t are distributed according to a skew normal distribution, where all values lie within three standard deviations of the mean ($\mu \pm 3\sigma$). This distribution has the same shape regardless of the number of folds (n) and other values of k also produce errors with similar distributions. Having observed a skew normal distribution for the instances of e_t, we now represent the accuracy of the corresponding regression model as an average of the individual values of e_t in the distribution (hereafter referred as e_a). Next, we determine the appropriate value of k, which controls how sparsely the training data is sampled. Figures 4.5 and 4.6 show the different values of e_a calculated for the regression models based on feature vectors f_1 through f_7 with values of k ranging from $1e-6$ to $1e-4$. As k increases (i.e., more observations are used to train the model), the regression model's accuracy (e_a) improves until k reaches $4e-5$, after which point it stabilizes. We thus use $k = 4e-5$ for training our models.

Another important point to note is that in our experiments, we fix the stratum size to 60 while performing stratified sampling, which means each stratum represents data from 60 consecutive time-steps of the RTM program. We fix the stratum size to limit the volume of

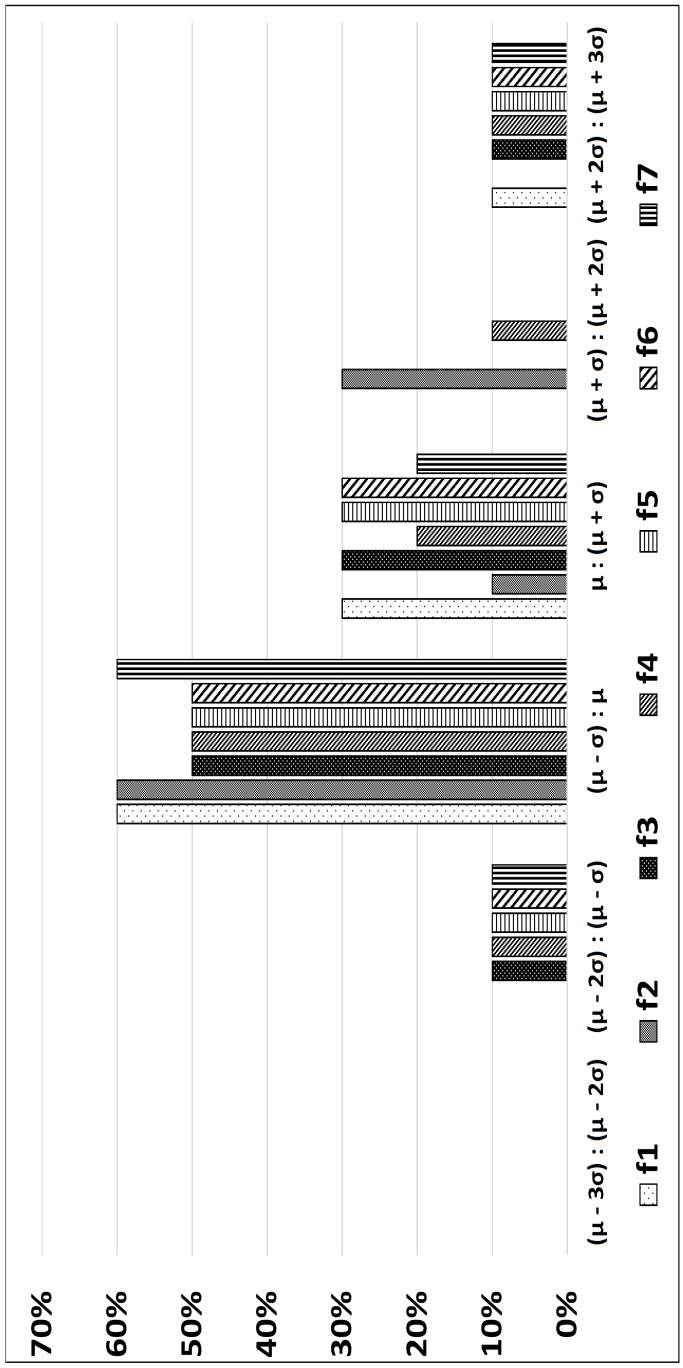

Figure 4.3: Distribution of e_t for 2-fold cross-validation

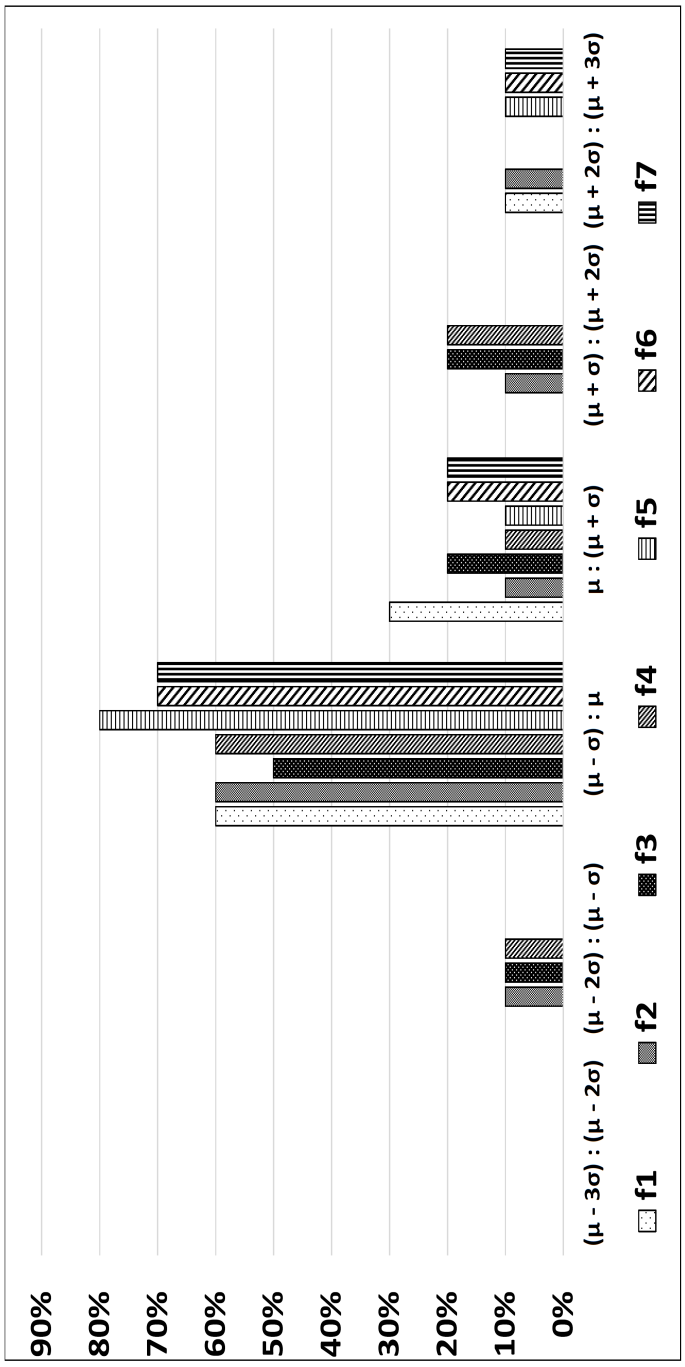

Figure 4.4: Distribution of e_i for 10-fold cross-validation

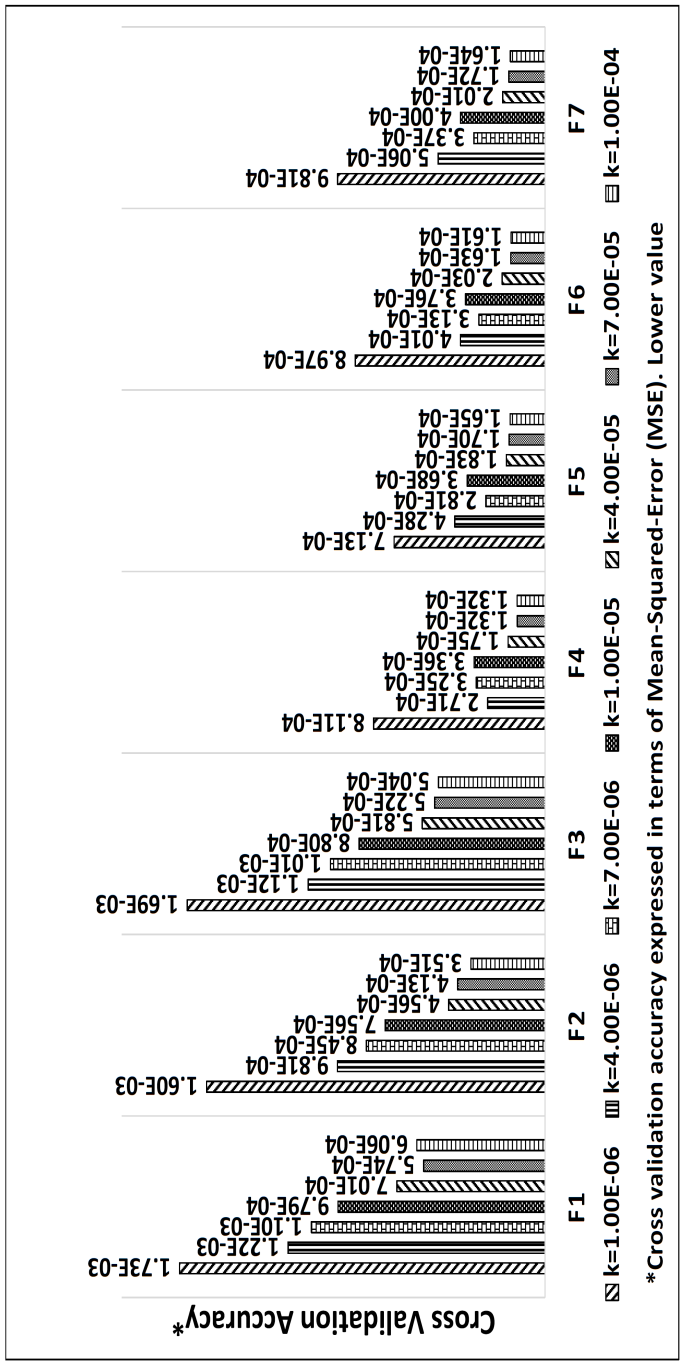

Figure 4.5: Sample size estimation using 2-fold cross-validation

Figure 4.6: Sample size estimation using 10-fold cross-validation

our experiments. In general, it is expected that the smaller the size of the stratum, the finer will be the representation of the individual time-steps, leading to a better cross-validation accuracy.

4.2.6 Error Model

We consider an error model involving a soft error occurring in a CPU's ALU operations and live register values. Further, we only consider a subset of the soft errors which cause SDC in the program output. Therefore, we do not consider all possible program locations in the RTM kernel for fault injections but rather only inject single-bit errors into a randomly-selected location in a three-dimensional array used by the stencil for computations. We perform fault injections at the application-level by instrumenting the RTM stencil kernel with a function call at the source level. The function is passed a live value stored at one of the locations of a three-dimensional array used by the RTM kernel. The function flips a single-bit at a random bit location of the value passed to it and returns the corrupted value. The array location whose value is chosen for fault injection is selected at random from the iteration space of the RTM kernel. Specifically, if an RTM kernel operating on a three-dimensional array with dimensions (x, y, z), and the number of time-steps for which it runs is t, then the iteration space of the RTM kernel is $x \times y \times z \times t$. This approach helps us to focus on our primary goal of studying the efficacy of our detectors in detecting SDC causing soft errors.

4.2.7 Threshold Selection and Error Detection Rate

To determine τ_{opt}, we plot ROC curves using the values of *true-positive* rate (R_{tp}) and *false-positive* rate (R_{fp}) computed by running two independent experiments. In the first experiment, we run the RTM kernel under each test input, and a bit-flip is injected during each run. We obtain the values for N_{tp} and N_{fn} from this experiment and use these values to compute R_{tp} using Equation 4.5. In the second experiment, we repeat all the steps followed in the first experiment without injecting any error. Using the result of the second experiment, we obtain N_{fp} and N_{tn} and use them to compute R_{fp} following Equation 4.6. We repeat these experiments for different threshold values for each of the detectors synthesized using feature vectors listed in Table 4.1. The threshold value chosen for our experiments starts with a small value, and is gradually increased until we achieve

reasonable *true-positive* and *false-positive* rates. Figures 4.7, 4.8, 4.9, 4.10, 4.11, 4.12, and 4.13 show high values of *true-positive* and *false-positive* rates (*top-right* area) for a low threshold value of 5. As the threshold value is increased in a fixed step-size of 5, the *true-positive* rate decreases at a much slower rate as compared to the *false-positive* rate, which demonstrates the effectiveness of these detectors. For a threshold value of 30, the number of *false-positives* comes down to zero while still providing a true-positive rate of $> 83\%$ for all the detectors.

Figure 4.14 presents the *true-positive* error detection rate and overhead of each detector. With a threshold value of $\tau_{opt} = 30$, we observe a high error-detection rate ($> 85\%$) with no false-positives for all the detectors. Detectors which use features f_1, f_2, f_3, and f_4 observe an average overhead of approximately 33%, whereas detectors with higher feature counts have an average overhead between 54% to 94%. This suggests that the former feature vectors are the best choice for making applications resilient to errors. Further, the fact that the addition of features between f_1 and f_4 has no effect on overhead suggests that the original cost may not be due to the actual computations that they perform, but rather other effects such as interference with the key computation's use of the memory hierarchy.

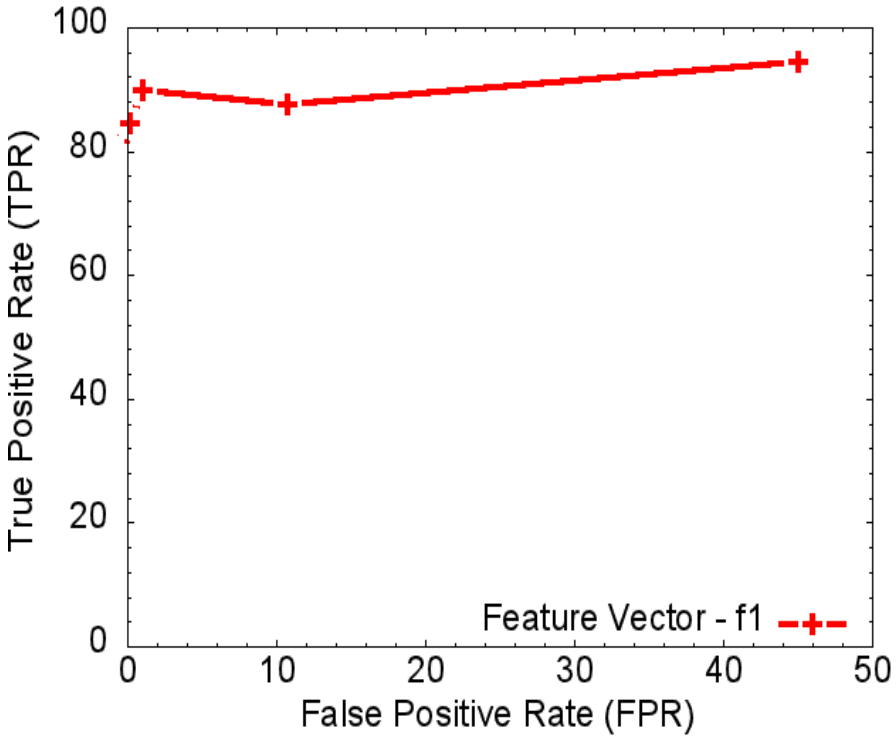

Figure 4.7: ROC curve for error detectors based on feature vector f_1

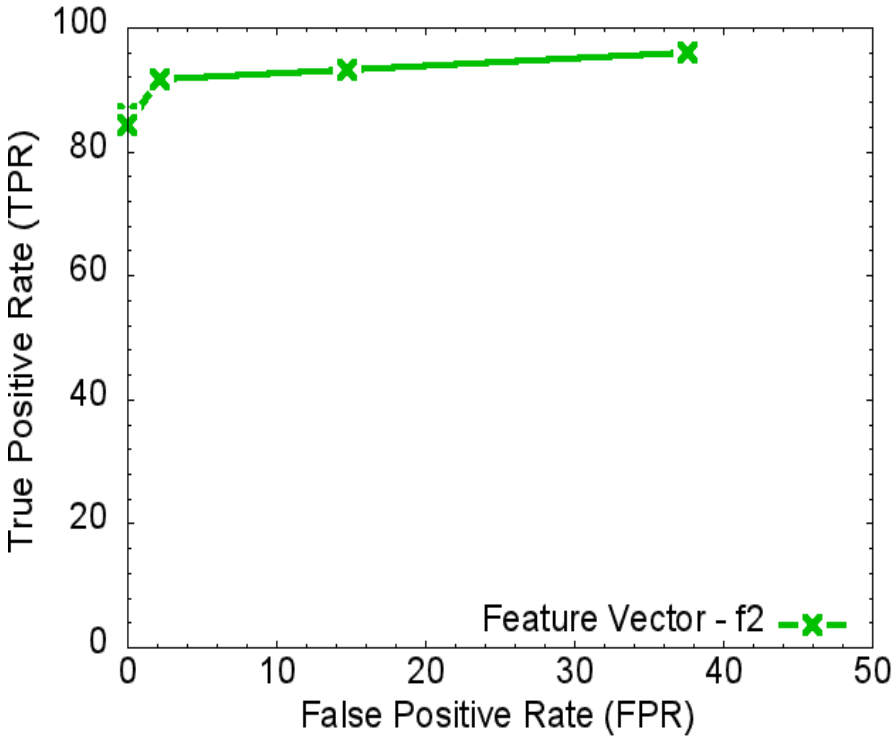

Figure 4.8: ROC curve for error detectors based on feature vector f_2

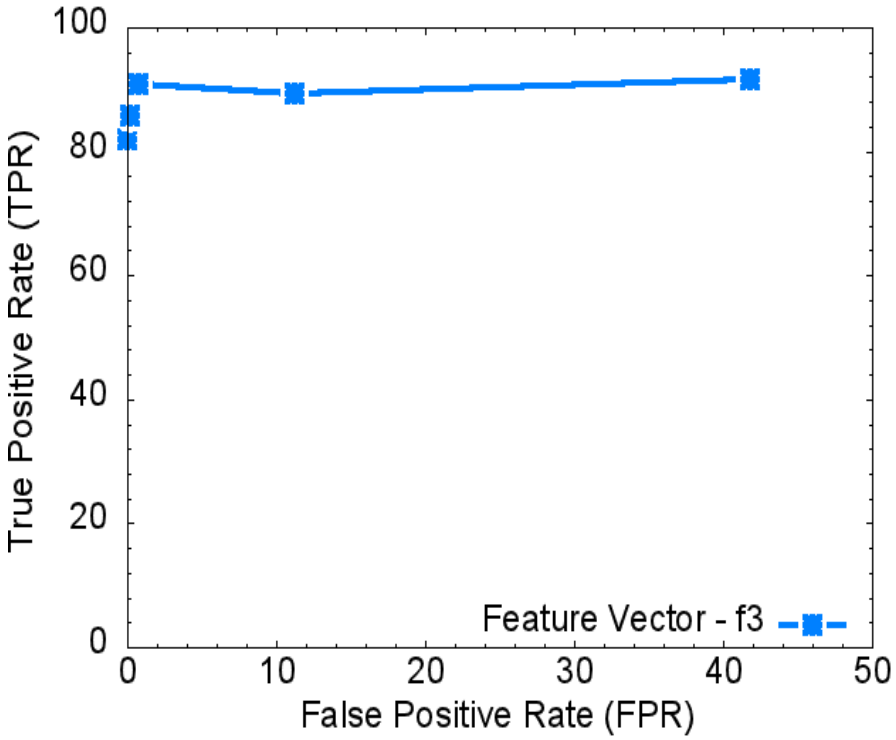

Figure 4.9: ROC curve for error detectors based on feature vector f_3

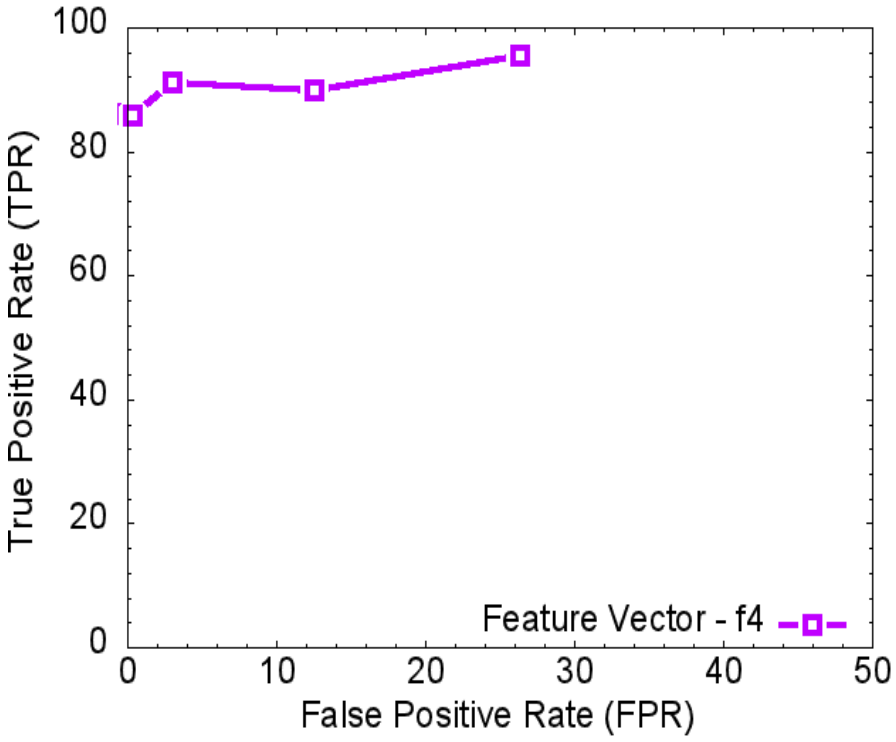

Figure 4.10: ROC curve for error detectors based on feature vector f_4

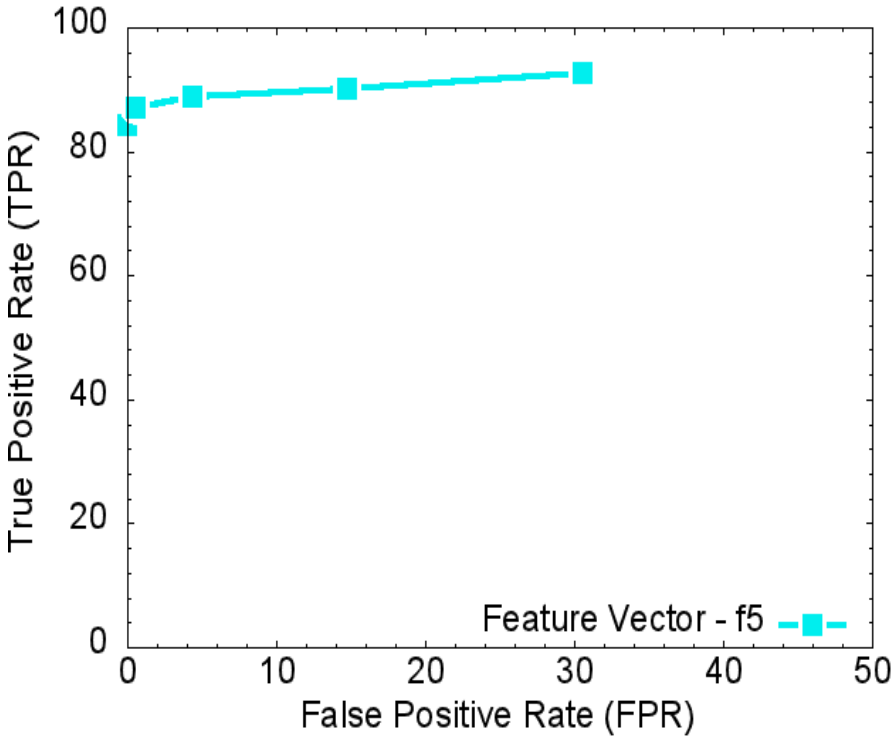

Figure 4.11: ROC curve for error detectors based on feature vector f_5

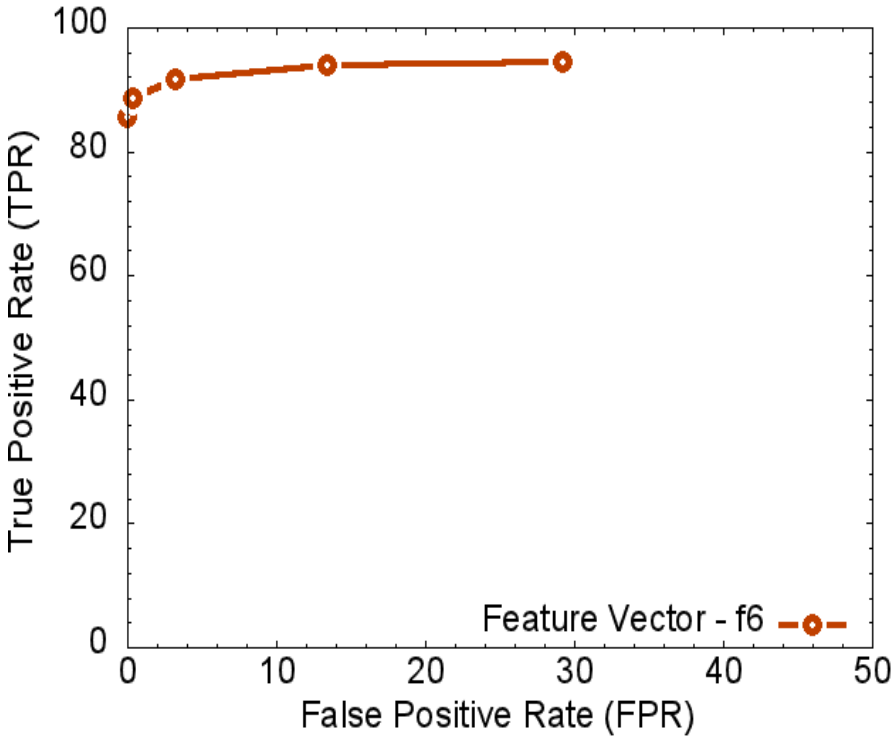

Figure 4.12: ROC curve for error detectors based on feature vector f_6

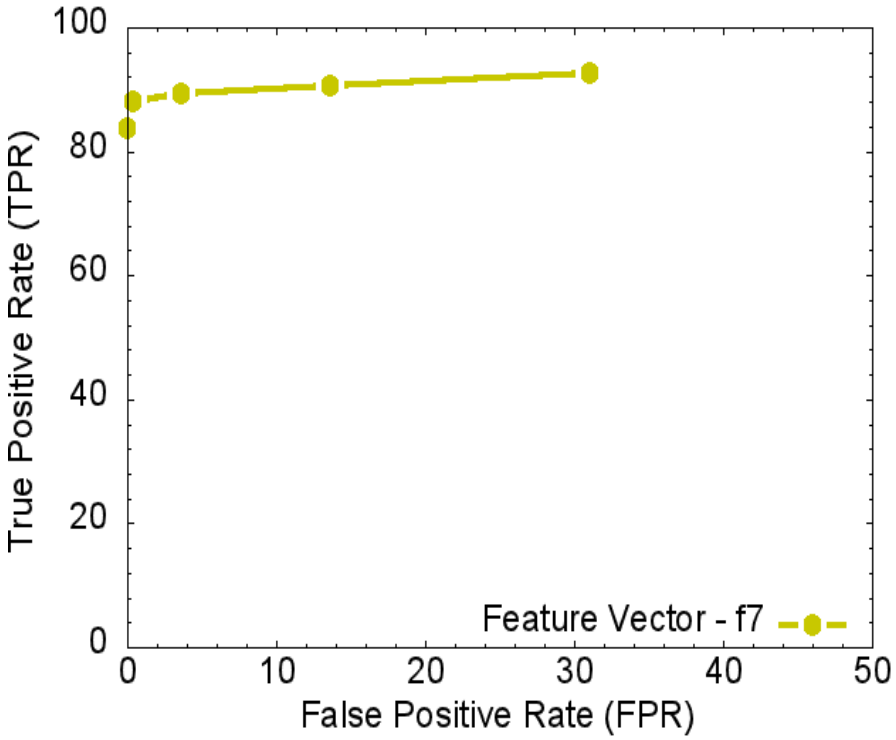

Figure 4.13: ROC curve for error detectors based on feature vector f_7

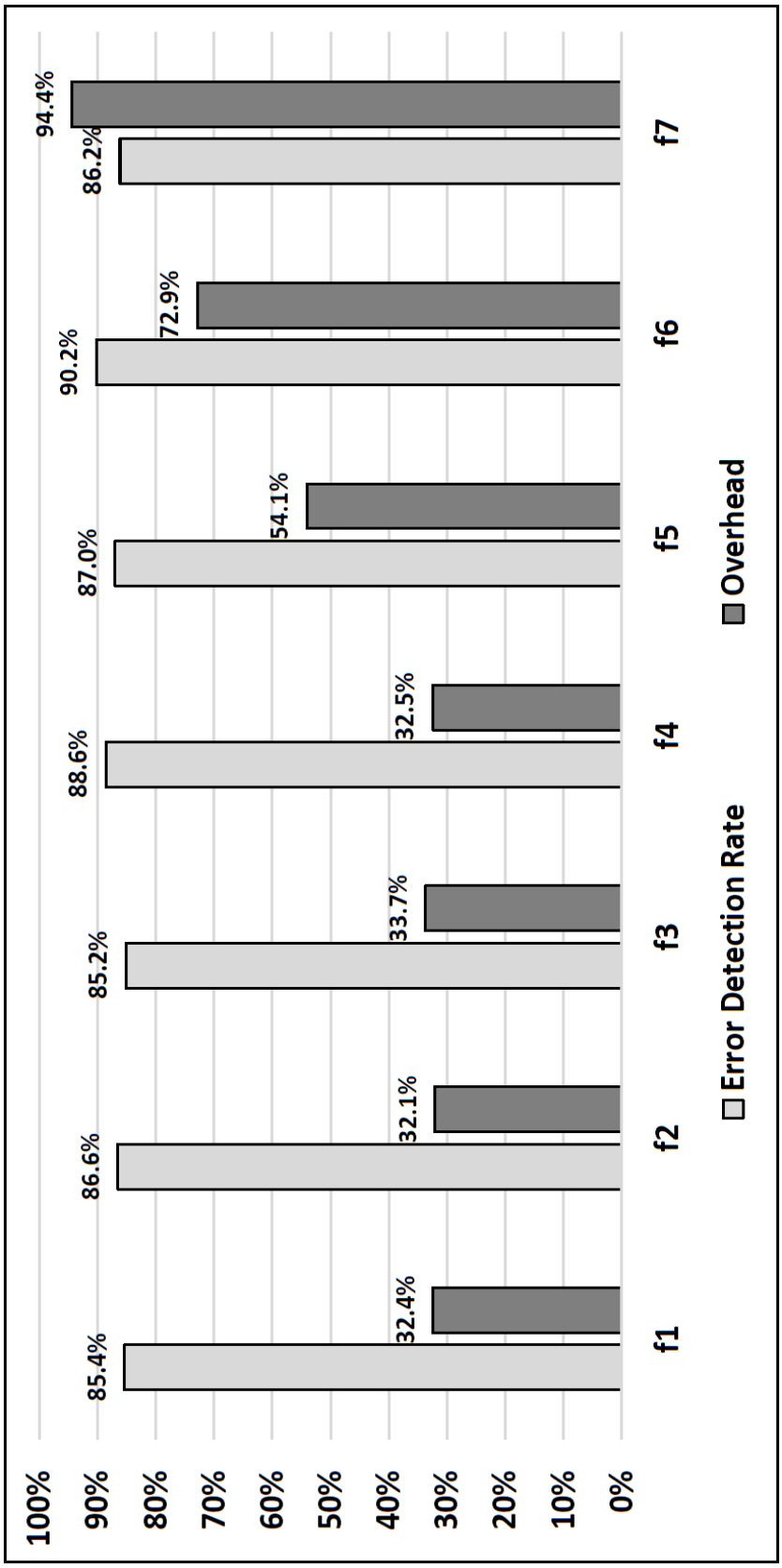

Figure 4.14: Error detection rate and overhead data with $\tau_{opt} = 30$

4.2.8 Solution of RTM Using Finite Difference Approximation

In this section, we derive a solution of the RTM PDE of Equation 4.2 which is eighth order accurate in space and second-order accurate in time. The purpose of this exercise is to demonstrate that finite-difference approximation gives fewer weight to the farther points on a stencil with respect to a point for which a new value has to be computed. To find an eighth order approximation of the second order derivative $P^{(2)}(x)$ in Equation 4.2, we consider *eight* nearby points $x+4h$,$x+3h$,$x+2h$,$x+h$,$x-h$,$x-2h$,$x-3h$, and $x-4h$. We express P at these points using a Taylor series expanded only up to eighth order derivative terms as shown by a set of equations in 4.7. Equation 4.7 can be presented as a set of linear equations in the form $Ax = B$ (assuming $h = 1$) and the solution can be presented in the form $x = A^{-1}B$ as shown in Equations 4.8 and 4.9, respectively. From Equation 4.9, we get second order derivate of $P(x)$ (shown in Equation 4.10) which is eighth order accurate.

$$
\begin{aligned}
P(x+4h) &= P(x) + 4hP^{(1)}(x) + \frac{(4h)^2}{2!}P^{(2)}(x) + ... + \frac{(4h)^7}{7!}P^{(7)}(x) + \frac{(4h)^8}{8!}P^{(8)}(x)\\
P(x+3h) &= P(x) + 3hP^{(1)}(x) + \frac{(3h)^2}{2!}P^{(2)}(x) + ... + \frac{(3h)^7}{7!}P^{(7)}(x) + \frac{(3h)^8}{8!}P^{(8)}(x)\\
P(x+2h) &= P(x) + 2hP^{(1)}(x) + \frac{(2h)^2}{2!}P^{(2)}(x) + ... + \frac{(2h)^7}{7!}P^{(7)}(x) + \frac{(2h)^8}{8!}P^{(8)}(x)\\
P(x+h) &= P(x) + hP^{(1)}(x) + \frac{h^2}{2!}P^{(2)}(x) + ... + \frac{h^7}{7!}P^{(7)}(x) + \frac{h^8}{8!}P^{(8)}(x)\\
P(x-h) &= P(x) + (-h)P^{(1)}(x) + \frac{(-h)^2}{2!}P^{(2)}(x) + ... + \frac{(-h)^7}{7!}P^{(7)}(x) + \frac{(-h)^8}{8!}P^{(8)}(x)\\
P(x-2h) &= P(x) + (-2h)P^{(1)}(x) + \frac{(-2h)^2}{2!}P^{(2)}(x) + ... + \frac{(-2h)^7}{7!}P^{(7)}(x) + \frac{(-2h)^8}{8!}P^{(8)}(x)\\
P(x-3h) &= P(x) + (-3h)P^{(1)}(x) + \frac{(-3h)^2}{2!}P^{(2)}(x) + ... + \frac{(-3h)^7}{7!}P^{(7)}(x) + \frac{(-3h)^8}{8!}P^{(8)}(x)\\
P(x-4h) &= P(x) + (-4h)P^{(1)}(x) + \frac{(-4h)^2}{2!}P^{(2)}(x) + ... + \frac{(-4h)^7}{7!}P^{(7)}(x) + \frac{(-4h)^8}{8!}P^{(8)}(x)
\end{aligned}
\tag{4.7}
$$

$$
\begin{pmatrix}
4 & \frac{(4)^2}{2!} & \frac{(4)^3}{3!} & \frac{(4)^4}{4!} & \frac{(4)^5}{5!} & \frac{(4)^6}{6!} & \frac{(4)^7}{7!} & \frac{(4)^8}{8!}\\
3 & \frac{(3)^2}{2!} & \frac{(3)^3}{3!} & \frac{(3)^4}{4!} & \frac{(3)^5}{5!} & \frac{(3)^6}{6!} & \frac{(3)^7}{7!} & \frac{(3)^8}{8!}\\
2 & \frac{(2)^2}{2!} & \frac{(2)^3}{3!} & \frac{(2)^4}{4!} & \frac{(2)^5}{5!} & \frac{(2)^6}{6!} & \frac{(2)^7}{7!} & \frac{(2)^8}{8!}\\
1 & \frac{1^2}{2!} & \frac{1^3}{3!} & \frac{1^4}{4!} & \frac{1^5}{5!} & \frac{1^6}{6!} & \frac{1^7}{7!} & \frac{1^8}{8!}\\
-1 & \frac{(-1)^2}{2!} & \frac{(-1)^3}{3!} & \frac{(-1)^4}{4!} & \frac{(-1)^5}{5!} & \frac{(-1)^6}{6!} & \frac{(-1)^7}{7!} & \frac{(-1)^8}{8!}\\
-2 & \frac{(-2)^2}{2!} & \frac{(-2)^3}{3!} & \frac{(-2)^4}{4!} & \frac{(-2)^5}{5!} & \frac{(-2)^6}{6!} & \frac{(-2)^7}{7!} & \frac{(-2)^8}{8!}\\
-3 & \frac{(-3)^2}{2!} & \frac{(-3)^3}{3!} & \frac{(-3)^4}{4!} & \frac{(-3)^5}{5!} & \frac{(-3)^6}{6!} & \frac{(-3)^7}{7!} & \frac{(-3)^8}{8!}\\
-4 & \frac{(-4)^2}{2!} & \frac{(-4)^3}{3!} & \frac{(-4)^4}{4!} & \frac{(-4)^5}{5!} & \frac{(-4)^6}{6!} & \frac{(-4)^7}{7!} & \frac{(-4)^8}{8!}
\end{pmatrix}
\begin{pmatrix}
P^{(1)}(x)\\ P^{(2)}(x)\\ P^{(3)}(x)\\ P^{(4)}(x)\\ P^{(5)}(x)\\ P^{(6)}(x)\\ P^{(7)}(x)\\ P^{(8)}(x)
\end{pmatrix}
=
\begin{pmatrix}
P(x+4)-P(x)\\ P(x+3)-P(x)\\ P(x+2)-P(x)\\ P(x+1)-P(x)\\ P(x-1)-P(x)\\ P(x-2)-P(x)\\ P(x-3)-P(x)\\ P(x-4)-P(x)
\end{pmatrix}
\tag{4.8}
$$

$$\begin{pmatrix} P^{(1)}(x) \\ P^{(2)}(x) \\ P^{(3)}(x) \\ P^{(4)}(x) \\ P^{(5)}(x) \\ P^{(6)}(x) \\ P^{(7)}(x) \\ P^{(8)}(x) \end{pmatrix} = \begin{pmatrix} 4 & \frac{(4)^2}{2!} & \frac{(4)^3}{3!} & \frac{(4)^4}{4!} & \frac{(4)^5}{5!} & \frac{(4)^6}{6!} & \frac{(4)^7}{7!} & \frac{(4)^8}{8!} \\ 3 & \frac{(3)^2}{2!} & \frac{(3)^3}{3!} & \frac{(3)^4}{4!} & \frac{(3)^5}{5!} & \frac{(3)^6}{6!} & \frac{(3)^7}{7!} & \frac{(3)^8}{8!} \\ 2 & \frac{(2)^2}{2!} & \frac{(2)^3}{3!} & \frac{(2)^4}{4!} & \frac{(2)^5}{5!} & \frac{(2)^6}{6!} & \frac{(2)^7}{7!} & \frac{(2)^8}{8!} \\ 1 & \frac{1^2}{2!} & \frac{1^3}{3!} & \frac{1^4}{4!} & \frac{1^5}{5!} & \frac{1^6}{6!} & \frac{1^7}{7!} & \frac{1^8}{8!} \\ -1 & \frac{(-1)^2}{2!} & \frac{(-1)^3}{3!} & \frac{(-1)^4}{4!} & \frac{(-1)^5}{5!} & \frac{(-1)^6}{6!} & \frac{(-1)^7}{7!} & \frac{(-1)^8}{8!} \\ -2 & \frac{(-2)^2}{2!} & \frac{(-2)^3}{3!} & \frac{(-2)^4}{4!} & \frac{(-2)^5}{5!} & \frac{(-2)^6}{6!} & \frac{(-2)^7}{7!} & \frac{(-2)^8}{8!} \\ -3 & \frac{(-3)^2}{2!} & \frac{(-3)^3}{3!} & \frac{(-3)^4}{4!} & \frac{(-3)^5}{5!} & \frac{(-3)^6}{6!} & \frac{(-3)^7}{7!} & \frac{(-3)^8}{8!} \\ -4 & \frac{(-4)^2}{2!} & \frac{(-4)^3}{3!} & \frac{(-4)^4}{4!} & \frac{(-4)^5}{5!} & \frac{(-4)^6}{6!} & \frac{(-4)^7}{7!} & \frac{(-4)^8}{8!} \end{pmatrix}^{-1} \begin{pmatrix} P(x+4) - P(x) \\ P(x+3) - P(x) \\ P(x+2) - P(x) \\ P(x+1) - P(x) \\ P(x-1) - P(x) \\ P(x-2) - P(x) \\ P(x-3) - P(x) \\ P(x-4) - P(x) \end{pmatrix} \tag{4.9}$$

$$\begin{aligned} P^{(2)}(x) = &-0.0017\{P(x+4) + P(x-4)\} + 0.0253\{P(x+3) + P(x-3)\} \\ &- 0.2\{P(x+2) + P(x-2)\} + 1.6\{P(x+1) + P(x-1)\} - 2.722P(x) \end{aligned} \tag{4.10}$$

$$P^{(2)}(n) = \{P(n+1) + P(n-1)\} - 2P(n) \tag{4.11}$$

Similarly, we can derive second order derivative of $P(n)$ which is second order accurate as shown in Equation 4.11. Equations 4.10 and 4.11 clearly show that the finite-difference approximation has a natural property of assigning fewer weight to the points which appear farther on a stencil. Intuitively, this suggests that higher-order stencils should be more amenable to spatial optimizations when deriving approximate kernel-based error detectors.

4.3 Related Work

Recently, there has been considerable interest in developing efficient error detectors for time-stepped stencil computations. The work by Benson et al. [10] proposes running a cost-effective but unstable solver alongside the main solver and declaring an error when the results of these solvers disagree beyond a given threshold. Our approach is very similar to this work, but we extract the secondary solver automatically using machine learning rather than selecting and implementing it manually for each main solver.

Another recent work by Berrocal et al. [11] uses regression functions to detect runtime anomalies caused due to soft errors. They use runtime execution data to learn these regression functions in an on-line fashion to detect anomalies. In contrast, in our approach, we derive a *less-expensive* approximate kernel of a stencil kernel as a redundant checker. Given that we derive the approximate function in a separate step in an off-line fashion,

it allows us to use more computational data for learning purposes without worrying about the potential implications on the detection overhead. This helps in maintaining the accuracy of the learned regression models without adversely impacting the detection overhead. Also, though not in the scope of this dissertation work, our methodology could also be used for gaining an understanding of large physical models by learning relations between inputs and outputs of these models.

Several other promising directions have been pursued towards building soft error detectors. While hardware and architecture-level protections [64, 66, 108] serve as a first line of defense, we must complement them with more flexible software-level techniques. Several software-level techniques employ control-flow-based detectors [72, 107], which rely on detecting illegal control transitions. These detectors are more suitable for control-flow rich applications, and less effective for data-intensive applications. Algorithm-based Fault Tolerance (ABFT) exploits algorithmic properties of a program to detect errors [34, 35, 101] but these solutions are problem-specific. In summary, to the best of our knowledge, none of the previous works employ methods for learning computationally less-expensive approximate kernels as redundant checkers for stencil computations, which is the key focus of our work.

4.4 Discussion

The error detection rates reported for the RTM kernel using approximate kernels are obtained by performing regression analysis on the inputs and the output of the RTM kernel when executed on a set of training inputs. The training inputs are obtained by randomizing parameters of an input distribution provided with the RTM kernel implementation [63] as shown in Figure 4.15. It is important to note that the prediction accuracy of the derived approximate kernels may or may not be the same for a different input distribution (e.g., normal distribution). However, the approach presented in this chapter can further be augmented by parameterizing input distributions as one of the dependent variables during the regression analysis phase. Similarly, one can also extract a cumulative distribution function (CDF) for the prediction accuracy of an approximate kernel for a given input distribution, and the extracted CDF can then be used for threshold estimation.

```
initialize_rtm_3d_array(){
   for( int z = 0; z < nz; ++z )
      for( int y = 0; y < ny; ++y )
         for( int x = 0; x < nx; ++x ){
             int i = z*ny*nx+y*nx+x;
             // randomize c1,c2
             float r = (x - nx/c1 + y - ny/c1 + z - nz/c1) / c2;
             r = max(c1-r, 0.0f) + c2;
             A[1][i] = A[0][i] = r;
         }
}
```

Figure 4.15: A pseudo code for initializing the RTM array

4.5 Conclusion

In this paper, we presented a novel approach using linear regression to efficiently approximate stencil kernels, validating it on the RTM stencil. We showed how the cost of training the model could be reduced via cross-validation-driven stratified sampling to reduce the training set size systematically. We also showed how to find a *near-optimal* threshold value(τ_{opt}) for the detectors using ROC curve analysis and presented the error detection rate and the overhead data for the detectors. A high error detection rate reported by our detectors demonstrates the effectiveness of our approach. We also showed that finite-difference approximation has a natural property of assigning fewer weight to farther points on a stencil. Therefore, in the case of higher-order stencils (such as the 25-point RTM stencil), we have a greater room for spatial optimization by using a smaller number of points on a stencil to build highly accurate error detectors. Alternatively, this also highlights one potential limitation of our approach that it may not be amenable to lower order stencils. In summary, through this exploratory work, we demonstrated that it is feasible to build efficient detectors by deriving lightweight approximations of higher-order stencil kernels using machine learning.

CHAPTER 5

PROTECTING STRUCTURED ADDRESS GENERATION FROM SOFT ERRORS

5.1 Introduction

High-performance computing (HPC) applications will soon be running at very high scales on systems with large component counts. The shrinking dimensions and reducing power requirements of the transistors used in the memory elements of these massively parallel systems make them increasingly vulnerable to temporary bit-flips induced by system noise or high-energy particle strikes. The temporary bit-flips occurring in memory elements are often referred to as soft errors. Previous studies project an upward trend in soft-error-induced vulnerabilities in HPC systems, thereby pushing down their mean-time-to-failure (MTTF) [18,19]. These trends drastically increase the likelihood of a bit-flip occurring in long-lived computations. Specifically, a bit-flip affecting computational states of a program under execution such as ALU operations or live register values may lead to silent data corruption (SDC) in the final program output. Making matters worse, such erroneous values may propagate to multiple compute nodes in massively parallel HPC systems [6].

The key focus of the work presented in this chapter is to detect bit-flips affecting address computation of array elements. For example, to load a value stored in an array A at an index `i`, a compiler must first compute the address of the location referred by the index `i`. A compiler performs this operation under-the-hood by using the base address of A and adding to it an offset value computed using the index `i`. This style of address generation scheme, which uses a base address and an offset to generate the destination address, is often referred to as the *structured address generation*. Accordingly, the computations done in the context of *structured address generation* are referred to as *structured address computations*.

Often, computational kernels used in HPC applications involve array accesses inside

loops, thus requiring *structured address computations*. For these kernels, there is a real chance of one of their *structured address computations* getting affected by a bit-flip. A *structured address computation* pertaining to an array, when subjected to a bit-flip, may produce an incorrect address that still refers to a valid address in the address space of the array. Using the value stored at this incorrect but *valid* address may lead to SDC without causing a program-crash or any other user-detectable-errors.

In this work, we demonstrate that the bit-flips affecting *structured address computations* for the above class of computational kernels lead to nontrivial SDC rates. Also, we present a novel technique for detecting bit-flips impacting *structured address computations*. Given that the *structured address computations* involve arithmetic operations that use a CPU's computational resources, we consider an error model where bit-flips affect ALU operations and CPU register files. We assume DRAM and cache memory to be error-free, a reasonable assumption because they are often protected using ECC mechanisms [23, 51, 52, 100]. We further limit the scope of our error model by considering only ALU operations and register values that correspond to *structured address computations*.

Specifically, we make the following contributions in this work:

1. We perform a fault injection driven study on ten benchmarks drawn from the PolyBench/C benchmark suite [83] demonstrating that *structured address computations* in these benchmarks when subjected to bit-flips, lead to nontrivial SDC rates.

2. We present a novel scheme that employs instruction-level rewriting of the address computation logic used in *structured address computations*. This rewrite *preserves* an error in a *structured address computation* by intentionally corrupting all *structured address computations* that follow it. This requires the creation of a dependency-chain between all *structured address computations* of a given array. The introduction of a dependency-chain enables the flow of error which helps in the following ways:

   (a) Strategic Placement of Error Detectors: Instead of checking each and every *structured address computations* for soft errors (which is prohibitively expensive), we strategically place our error detectors at the end of a dependency-chain.

   (b) Promoting SDCs to Program-crashes: By enabling the flow of error in address computation logic, we increase the chances of promoting an SDC to a program-crash (that is more easily detected).

3. We present a methodology for implementing our proposed scheme as a compiler-level technique called PRESAGE (**PR**ot**E**cting **S**tructured **A**ddress **GE**neration). Specifically, we have implemented PRESAGE using LLVM compiler infrastructure [55, 58] as a transformation pass. LLVM preserves the pointer-related information at LLVM intermediate representation (IR) level (as also highlighted in recent works [69, 110]) while providing access to a rich set of application programming interfaces (APIs) for seamlessly implementing PRESAGE transformations. This is the key reason behind choosing LLVM as the tool-of-choice.

In summary, our error-detection approach is based on the following principle:

The larger the fraction of system state an error corrupts, the easier it is to detect.

The rest of the chapter is organized as follows. Section 5.2 explains the key idea through a set of small examples. Section 5.3 formally introduces the key concepts and the methodology used to implement PRESAGE. In Section 5.4, we provide a detailed analysis of the experiments carried out to measure the efficacy of PRESAGE. Section 5.5 provides a literary review of the closely related work done in this area. Section 5.6 discusses the pros and cons of the PRESAGE technique. Finally, Section 5.7 summarizes the key takeaways and future directions for this work.

5.2 Motivating Example

Figure 5.1 presents a simple C function `foo1` performing store operations to even-indexed memory locations of an array `a[]` of size `2n` inside a `for` loop. It also stores the last accessed array address into a variable `addr` at the end of every loop-iteration. Figure 5.2 represents the corresponding x86 code emitted for the `foo1` function when compiled using clang compiler with `O1` optimization level. Registers `%esi` and `%ecx` represent the variable `n` and the loop iterator `i` of the function `foo1`, whereas registers `%rdi` and `%rax` correspond

```
L0: void foo1(double* a, unsigned n){
L1:   double* addr=a;
L2:   for(int i=1;i<n;i++){
L3:     int id=2*i-2;
L4:     addr=&a[id];
L5:     *addr=i;
      }
  }
```

Figure 5.1: An example function foo1()

```
L0:  cmp      0x2,%esi
L1:  jl       L12
L2:  xor      %eax,%eax
L3:  mov      0x1,%ecx
L4:  xorps    %xmm0,%xmm0
L5:  cvtsi2sd %ecx,%xmm0
L6:  cltq
L7:  movsd    %xmm0,(%rdi,%rax,8)
L8:  add      0x2,%eax
L9:  inc      %ecx
L10: cmp      %ecx,%esi
L11: jne      L4
L12: retq
```

Figure 5.2: An x86 representation of the example function foo()

to the array's base address and index, respectively. In every loop iteration, a destination array address is computed by the expression `(%rdi,%rax,0x8)` which evaluates to `(0x8*%rax +%rdi)`, the value in register `%rax` is incremented by 2, and the base address stored in `%rdi` remains fixed. It is worth noting that the final address computation denoted by the expression `(%rdi,%rax,0x8)` is *not user-visible* and is something the compiler does under-the-hood.

In contrast to the fixed base address (FBA) scheme used in function `foo1`, function `foo2` (shown in Figure 5.3), a semantically equivalent version of `foo1`, introduces a novel relative base address (RBA) scheme. Specifically, `foo2` uses an array address computed in a loop iteration (`addr`) as the new base address for the next loop iteration along with a relative index (`rid`) as shown in Figure 5.1. This simple but powerful scheme creates a dependency-chain in the address computation logic as the computation of any new address would depend on the last computed address. Therefore, our RBA scheme guarantees that if an address computation of an array element gets corrupted, then all subsequent address computations would also become erroneous. This, in turn, enables us to strategically place error detectors at a handful of places in a program (preferably at all program exit points), thereby making the whole error detection process lightweight. For example, in functions `foo1` and `foo2`, the address of a new array element, computed during every loop iteration, is stored in the variable `addr`. The value stored in the variable `addr` may get corrupted in following scenarios:

- **Error Scenario I**: A bit-flip occurs in the value stored in the loop-iterator variable `i` in functions `foo1` and `foo2`.

```
L0: void foo2(double *a, int n){
L1:   double* addr=a;
L2:   int pid=0;
L3:   for(int i=1;i<n;i++){
L4:     int id=2*i-2;
L5:     int rid=id-pid;
L6:     addr[rid]=i;
L7:     pid=id;
L8:     addr=&addr[rid];
      }
   }
```

Figure 5.3: An example function foo2()

- **Error Scenario II**: A bit-flip affects the value stored in the absolute index variable `id` in functions `foo1` and `foo2`.

- **Error Scenario III**: A bit-flip occurs in the value stored in the relative index variable `rid`, which is only present in the function `foo2`.

- **Error Scenario IV**: A bit-flip affecting the value stored in the variable `addr` in functions `foo1` and `foo2`.

The above program-level sites are listed in Table 5.1 for easy reference. With respect to the error scenario IV, it is evident that only in the case of `foo2`, when the result of final address computation stored in `addr` is corrupted during one of the loop iterations, all subsequent address computations in the remaining loop iterations would also get corrupted due to the dependency-chain introduced in the address computation logic. We further demonstrate the behavior of these dependency-chains, introduced by our RBA scheme, through a small set of fault injection driven experiments. Figure 5.4 presents the result of two independent runs for function `foo1`. The X-axis shows the number of loop iteration whereas the Y-axis

Table 5.1: List of fault sites in functions `foo1` and `foo2`

Fault Site	Description
`i`	Loop iterator variable.
`id`	Absolute index variable.
`addr`	A variable containing an address of a location in the array `a[]`.
`rid`	Relative index variable (only present in `foo2`).

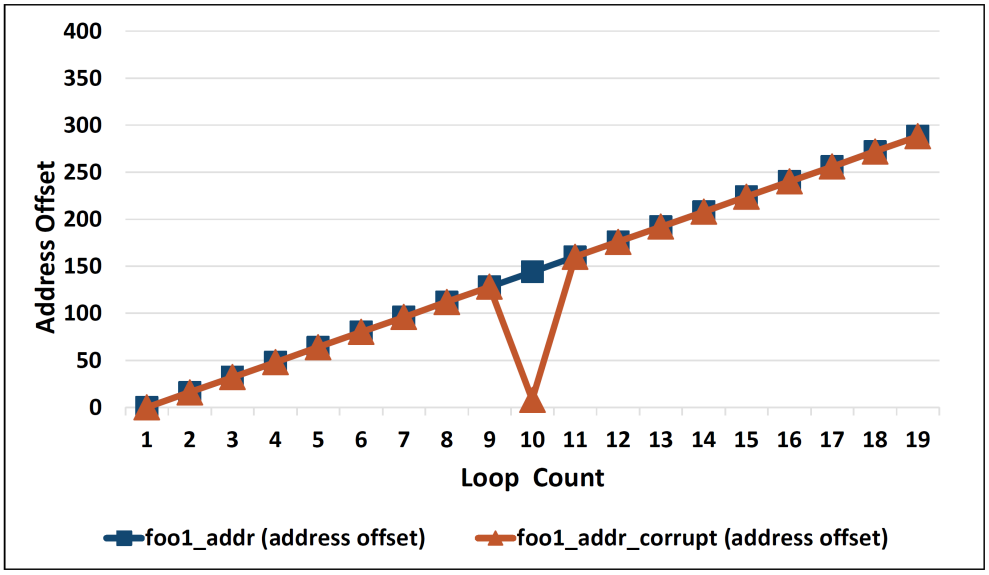

Figure 5.4: Function `foo1` with no dependency-chains

shows the value stored in the variable `addr`. The execution with label `foo1_addr` represents a fault-free execution of `foo1`.

The execution with label `foo1_addr_corrupt` represents a faulty execution of `foo1` where a single bit fault is introduced at bit position 6 of the value stored in `addr` during the tenth loop iteration. Similarly, Figure 5.5 presents the result of two independent runs for function `foo2` such that a single bit fault is introduced at bit position 6 of the value stored in `addr` during the first loop iteration in the faulty execution represented by the label `foo2_addr_corrupt`. We can clearly notice that only in the case of function `foo2`, once an address value stored in `addr` gets corrupted, all subsequent address values stored in `addr` are also corrupted.

5.3 Methodology

Section 5.2 demonstrates that a simple rewrite of the address computation logic introduces a dependency-chain, thereby enabling the flow of error. Given that the address computation is often done in a user-transparent manner by the compiler, we implement our technique at the compiler-level. Specifically, we choose the LLVM compiler infrastructure to implement our technique as a transformation pass (here on referred to as PRESAGE) which works on LLVM's intermediate representation (IR).

Our implementation eliminates the need for any manual effort from programmers,

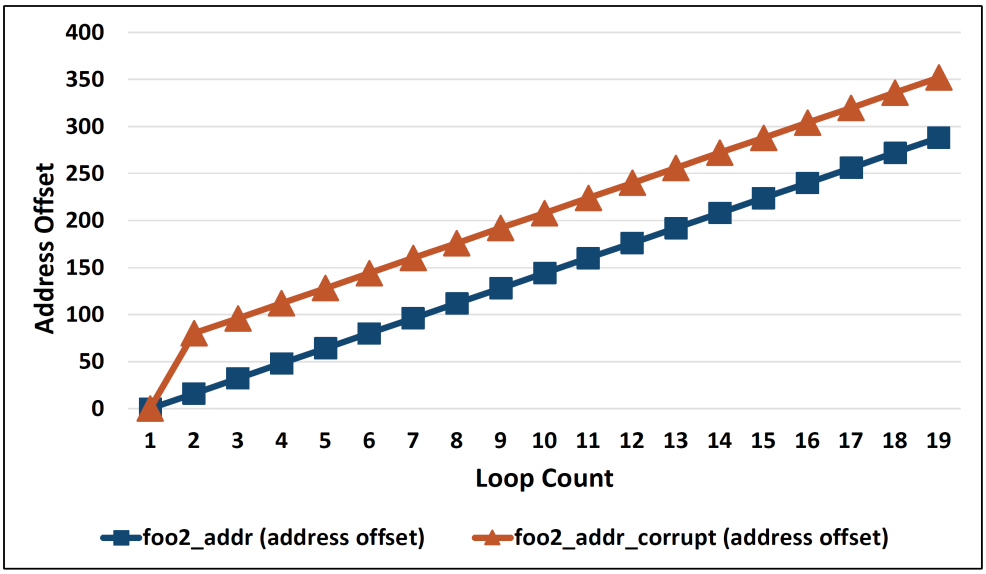

Figure 5.5: Function `foo2` with a dependency-chain introduced

thereby allowing our technique to scale to nontrivial programs. LLVM's intermediate representation (IR) provides a special instruction called `getelementptr` (here on referred as `GEP` for brevity) for performing address computation of `Aggregate` types including `Array` type[1]. Therefore, all analyses implemented as part of PRESAGE are centered around the `GEP` instruction. Table 5.2 presents the glossary of terms that are referred to in the remaining sections of this chapter. A `GEP` instruction requires a base address, one or more index values, and the size of an element to compute an address, often referred to as *structured address*. Given the key focus of our work is to protect these *structured addresses*, the definition of an array on which PRESAGE transformations are applied closely follows the LLVM's `Array` type definition with some restrictions as explained below:

- ***Definition 1***: An *array* in this work always refers to a contiguous arrangement of elements of the same *type* laid out linearly in the memory.

- ***Definition 2***: All structured address computations protected using PRESAGE must always use only one index for address computations. It is important to note that this is needed only to simplify the implementation and does not limit the scope of PRESAGE as multidimensional arrays can be easily represented using single-

[1]LLVM's type system is explained in its language reference manual located at: `http://llvm.org/docs/LangRef.html`

Table 5.2: Terminologies

Term	Description
$\mathcal{F}$	A target function on which PRESAGE transformations are applied.
$\mathbf{b}$	A base address with at least one user in the target function.
$\mathcal{B}$	A basic block in the target function.
$E(\mathcal{B}_1, \mathcal{B}_2)$	A boolean function which returns *true* only if an edge exists from $\mathcal{B}_1$ to $\mathcal{B}_2$.
$\mathcal{L}_{\mathcal{B}_p}(\mathcal{B})$	A set of all immediate predecessor basic blocks of $\mathcal{B}$.
$\mathcal{L}_{\mathcal{B}_s}(\mathcal{B})$	A set of all immediate successor basic blocks of $\mathcal{B}$.
$\mathcal{L}_{\mathcal{B}_e}(\mathcal{F})$	A set of all exit basic blocks in the target function $\mathcal{F}$.
$\mathcal{L}_{\mathbf{b}}(\mathcal{F})$	A set of all immutable base addresses in $\mathcal{F}$.
$\mathcal{L}_{\mathcal{G}}(\mathcal{B}, \mathbf{b})$	A set of all `GEP` instructions in $\mathcal{B}$ which use the base address $\mathbf{b}$.
$\mathcal{M}_{\phi}$	A two-level nested hashmap with first key a basic block, second key a base address mapped to a `phi` node.
$\mathcal{M}_{\mathcal{G}}$	A two-level nested hashmap with first key a basic block, second key a base address mapped to a `GEP` instruction.

indexed scheme. For example, a two-dimensional array could be laid out linearly in memory by traversing it in row-major or column-major fashion.

- ***Definition 3***: The base addresses used in all structured address computations protected by PRESAGE must not be updated. For example, if a PRESAGE transformation is applied to a callee function to protect its structured address computations, then the callee function must not mutate the base addresses referenced in structured address computations protected by PRESAGE.

5.3.1 Error Model

We consider an error model where soft errors induce a single-bit fault affecting CPU register files and ALU operations. We assume that memory elements such as data cache and DRAM are error-free because they are usually protected using ECC mechanisms. We implement our error model by targeting runtime instances of LLVM IR-level instructions of a target function for fault injection.

For example, if there are N dynamic IR-level instructions observed corresponding to a

target function, then we choose one out of N dynamic instructions with a uniform random probability of $\frac{1}{N}$ and flip the value of a randomly chosen bit of the destination virtual register, i.e., the left-hand side of the randomly chosen dynamic instruction. Similar error models have been proposed in the past for various resilience studies and it provides a reasonable estimate of the application-level resiliency of an application [59, 97]. Given that our focus is to study soft errors affecting *structured address computation*, we consider all fault sites which when subjected to a random single-bit bit-flip may affect the output of one or more `GEP` instructions of a target program. Specifically, we propose the two following error models which mainly differ in the dynamic fault site selection strategy.

5.3.1.1 Error Model I

As described in Section 5.2, error scenarios affecting *structured addressed computations* are broadly categorized into soft errors affecting *index values* and the final output of `GEP` instructions. Error model I considers the scenario where *index values* are corruption-free, but the final output of one of the `GEP` instruction has a random single-bit corruption. This is done by randomly choosing from dynamic instances of all GEP instructions of a target function and injecting a bit-flip in the final address computed the `GEP` instruction.

5.3.1.2 Error Model II

Error model II considers the case where the *index value* of one of the dynamic instances of GEP instructions is corrupted including the dynamic fault sites corresponding to the set of *def-use* leading to the *index-value*.

The above two error models are implemented using an open-source and publicly available fault injector tool VULFI [4, 95]. Also, note that in our error models, we do not target base addresses as these are small in numbers (one per array) and can be easily protected through replication without incurring severe performance or space overhead.

5.3.2 PRESAGE Transformations

We refer to two or more `GEP` instructions as *same-class* `GEP`s if they use the same base address. PRESAGE creates a dependency-chain between *same-class* `GEP`s in a two-stage process.

5.3.2.1 Inter-Block Dependency Chains

The first stage involves enabling dependency-chains between *same-class* `GEP`s in different basic blocks. Intuitively, it would require first `GEP`, for a given base address, appearing in all basic blocks be transformed in a manner such that it uses the address computed by the last *same-class* `GEP` in its predecessor basic block as the *relative base*. However, we need a bit more careful analysis as a basic block may have more than one predecessor basic blocks. Moreover, it might be possible that not all predecessor blocks have a *same-class* GEP or a predecessor block might be a *back edge* (i.e., there is a loop enclosing the basic block and its predecessor basic block). Therefore, we propose a three-step process for linking *same-class* `GEP`s in different basic blocks as explained by Figures 5.6 and 5.7. As a first step, as shown in Figure 5.6, we iterate over all basic blocks of a target function $\mathcal{F}$ in a breadth-first order. In a given basic block $\mathcal{B}$ with an incoming edge count e, we insert a `phi` node for each unique base address appearing in $\mathcal{L}_{\mathbf{b}}(\mathcal{F})$ for selecting a value from *same-class* incoming `GEP` values (each belonging to a unique predecessor basic block). For a given base address **b**, the respective `phi` node entry is used as the *relative base* by the first `GEP` (with base **b**) in the current basic block $\mathcal{B}$. In case $\mathcal{B}$ does not have a valid `GEP` entry for **b** , then we call $\mathcal{B}$ a *pass-through* basic block with respect to **b**. In this case, we simply pass the `phi` node value to the successor basic blocks. We use a `phi` node because all PRESAGE transformations are applied at LLVM IR and LLVM uses the single static assignment (SSA) form, thus requiring a `phi` node to select a value from one or more incoming values. For

```
1:  CreateInterBlkDepChain(F,M_G,M_φ) {
2:      for all B in BFS(F) {
3:          e ← GetIncomingEdgeCount(B)
4:          for all b in L_b(F) {
5:              φ ← CreateEmptyPHINode(b,e)
6:              InsertPHINodeEntry(B,b,φ,M_φ)
7:              for all B_p in L_Bp(B) {
8:                  if HasGEP(B_p,b,M_G) {
9:                      G ← GetGEP(B_p,b,M_G)
10:                     SetIncomingEdge(B_p,B,φ,G)
11:                 }
12:             }
13:         }
14:     }
15: }
```

Figure 5.6: PRESAGE algorithm for creating inter-block dependency-chains

```
1:  UpdateInterBlkDepChain(F,M_φ,M_G,P) {
2:      for all B in BFS(F) {
3:          for all B_p in L_{B_p}(B) {
4:              for all b in L_b(F) {
5:                  s ← ¬HasGEP(B_p,b,M_G)
6:                  s ← s ∧ HasPHI(B_p,b,M_φ)
7:                  s_1 ← s ∧ ¬IsBackEdge(B_p,B)
8:                  s_1 ← s_1 ∧ (P = Pass1)
9:                  s_2 ← s ∧ IsBackEdge(B_p,B)
10:                 s_2 ← s_2 ∧ (P = Pass2)
11:                 if s_1 ∨ s_2 {
12:                     φ ← GetPHINode(B,b,M_φ)
13:                     φ_p ← GetPHINode(B_p,b,M_φ)
14:                     SetIncomingEdge(B_p,B,φ,φ_p)
15:                 }
16:             }
17:         }
18:     }
19: }
```

Figure 5.7: PRESAGE algorithm for updating inter-block dependency-chains

each `phi` node entry created in $\mathcal{B}$, if valid incoming `GEP` values are available from one or more predecessor basic blocks, the `phi` node is updated with those values by calling the *SetIncomingEdge* routine. At this point, we already have created `phi` node entries in each basic block (including all *pass-through* basic blocks), and have populated these `phi` nodes with incoming `GEP` values wherever applicable. As the next step, as shown in Figure 5.7, for a basic block $\mathcal{B}$ with each of its *pass-through* predecessor basic block with respect to a base address **b**, the respective `phi` node ϕ is updated with the predecessor's `phi` node entry ϕ_p by calling *SetIncomingEdge* routine. If a back-edge exists from a *pass-through* predecessor basic block $\mathcal{B}_p$ to $\mathcal{B}$ (i.e., there exists a loop enclosing $\mathcal{B}$ and $\mathcal{B}_p$) then $\mathcal{B}$ may receive invalid data from $\mathcal{B}_p$ as $\mathcal{B}_p$ is also a successor basic block of $\mathcal{B}$. Therefore, we invoke the procedure *UpdateInterBlkDepChain* in Figure 5.7 twice. In the first pass, the `phi` node entries of all *pass-through* predecessor basic blocks of $\mathcal{B}$, which do not have back edges to $\mathcal{B}$, are assigned to the respective `phi` node entries in $\mathcal{B}$. In the second pass, we repeat the steps of the first pass with the exception that this time, we select the `phi` node entries of all *pass-through* predecessor basic blocks of $\mathcal{B}$ which do have back edges to $\mathcal{B}$.

5.3.2.2 Intra-Block Dependency Chains

The second stage involves creating intra-block dependency-chains. As shown in Figure 5.8, for each basic block $\mathcal{B}$ of a target function $\mathcal{F}$ and for each unique base address $\mathbf{b} \in \mathcal{L}_{\mathbf{b}}(\mathcal{B})$, if there exist one or more *same-class* `GEP` instructions which use $\mathbf{b}$ as the base, we need to transform these `GEP`s to create a dependency-chain. In other words, each `GEP` uses the value computed by the previous `GEP` as the *relative base* using our RBA scheme. For the first occurrence of `GEP` instruction in $\mathcal{B}$ with base $\mathbf{b}$, we extract the *relative base* information using the *phi* node entry ϕ created in the previous stage. At runtime, the `phi` node ϕ will receive the last address computed using the base address $\mathbf{b}$ from one of the predecessor basic blocks of $\mathcal{B}$. In summary, for each `GEP` instruction $\mathcal{G}$, an equivalent version $\mathcal{G}_n$ is created using the *relative base* and the *relative index* values. All uses of $\mathcal{G}$ are then replaced by $\mathcal{G}_n$ and $\mathcal{G}$ is then finally deleted.

5.3.3 Detector Design

The error detectors are designed to protect against single-bit faults injected using error model I. As shown in Figure 5.9, in each exit basic block $\mathcal{B}_e$, for each unique base address $\mathbf{b}$, PRESAGE makes available the value computed of the last run `GEP` instruction with base $\mathbf{b}$ and the *relative index* value used. Additionally, PRESAGE also makes available the *absolute index* value which along with the base address $\mathbf{b}$ can also be used to reproduce the output of the last run `GEP` instruction with base $\mathbf{b}$. The error detectors then simply check if the output $\mathcal{G}$ produced by the last run `GEP` instruction matches the recomputed value $\mathcal{G}_d$ using the base address $\mathbf{b}$ and the *absolute index* value. Given that in error model I, we consider the base address and index value to be corruption free, the error detectors are *precise* with respect to error model I as they do not report any *false positives*.

Figure 5.10 shows the LLVM-level control-flow graph (CFG) of the function `foo1` presented in Section 5.2. Similarly, Figure 5.11 shows the LLVM-level CFG of the PRESAGE transformed version of the function `foo1`. The `GEP` instruction in function `foo1` (Figure 5.10) which stores the computed address in register `%13` is replaced by a new `GEP` instruction (Figure 5.11) in the PRESAGE transformed version of `foo1` which uses relative base and relative index value for address computation. The PRESAGE transformed version of `foo1` in Figure 5.10 also has error detector code inserted in the exit basic block. Specifically, `%GEP_duplct` represents the recomputed version of the address which is com-

```
1:  CreateIntraBlkDepChain(F,M_G,M_φ) {
2:      for all B in BFS(F) {
3:          for all b in L_b(F) {
4:              for all G in L_G(B, b) {
5:                  if IsFirstGEP(G) {
6:                      φ ← GetPHINode(B,b,M_φ)
7:                      b_r ← GetRelativeBase(φ,b)
8:                      pid ← GetPrevIdx(φ)
9:                  }
10:                 id ← GetCurrentIdx(G)
11:                 rid ← GetRelativeIdx(id,pid)
12:                 G_n ← CreateNewGEP(γ,rid)
13:                 pid ← id
14:                 InsertGEP(G_n,G)
15:                 ReplaceAllUses(G,G_n)
16:                 DeleteGEP(G)
17:             }
18:         }
19:     }
20: }
```

Figure 5.8: PRESAGE algorithm for creating intra-block dependency-chains

```
1:  InsertDetectors(F,M_G,M_φ) {
2:      for all B_e in L_{B_e}(F) {
3:          for all b in L_b {
4:              φ ← GetPHINode(B,b,M_φ)
5:              b_r ← GetRelativeBase(φ,b)
6:              rid ← GetRelativeIdx(φ)
7:              pid ← GetPrevIdx(φ)
8:              G ← CreateNewGEP(b_r,rid)
9:              G_d ← CreateNewGEP(b,pid)
10:             InsertEqvCheck(G,G_d)
11:         }
12:     }
13: }
```

Figure 5.9: PRESAGE algorithm for error detection

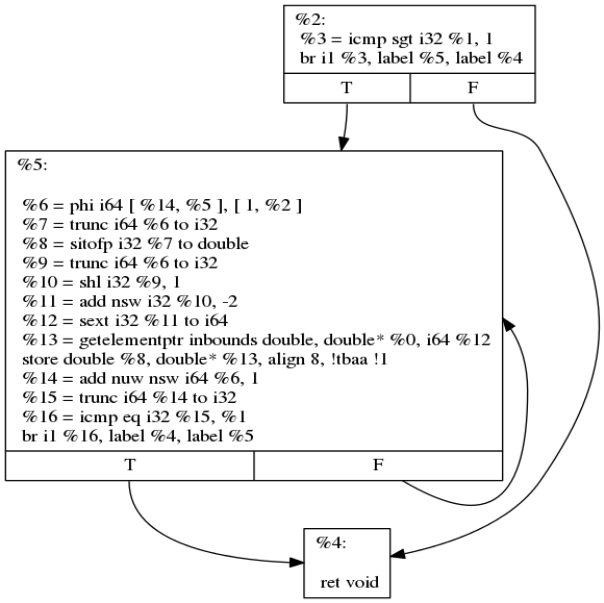

Figure 5.10: CFG representation of the function `foo1`

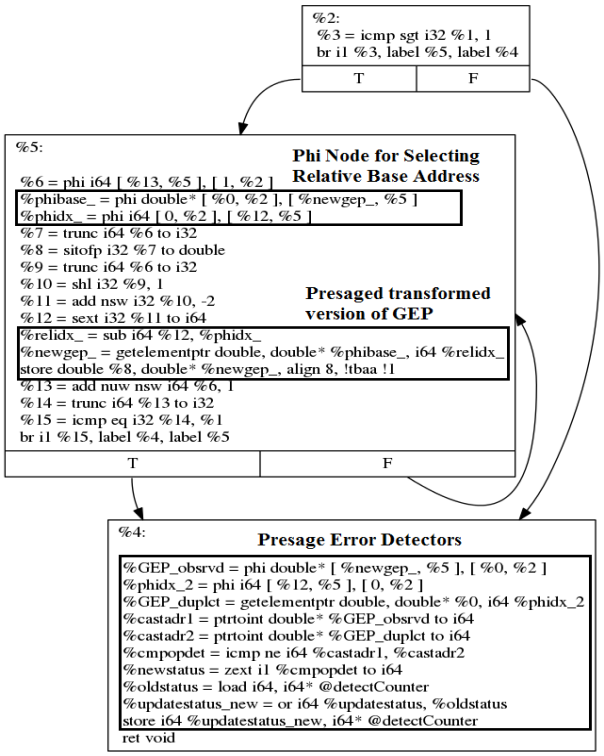

Figure 5.11: CFG representation of PRESAGE transformed version of the function `foo1`

pared against the observed address value `%GEP_obsrvd`. In case of a mismatch, the global variable `@detect-Counter` is set to report error detection to the end user.

5.4 Experimental Results

5.4.1 Evaluation Strategy

Our evaluation strategy involves measuring the effectiveness of the proposed error detectors in terms of SDC detection rate and performance overhead. Also, we analyze the impact of PRESAGE transformations on an application's resiliency using a fault injection driven study. We consider ten benchmarks (listed in Table 5.3) drawn from the PolyBench/C benchmark suite [82].

These benchmarks represent a diverse set of applications from areas such as stencils, algebraic kernels, solvers, and BLAS routines. For each of these benchmarks, we perform four set of experiments, summarized in Table 5.4. Each experiment set involves 100 fault

Table 5.3: List of benchmarks

Benchmark	Avg. DIC-I (in millions)	Avg. DIC-II (in millions)	SIC-I	SIC-II	Total SIC	%SI-I	%SI-II
adi	59.2	157.5	30	69	161	18.6%	42.8%
fdtd-2d	63.7	24.8	68	98	249	27.3%	39.3%
seidel-2d	74.8	36.8	42	114	180	23.3%	63.3%
jacobi-2d	64.2	97.1	56	112	196	28.5%	57.1%
gesummv	0.4	0.7	5	5	22	22.7%	22.7%
trmm	39.1	107.1	14	39	90	15.5%	43.3%
atax	0.5	0.7	22	26	91	24.1%	28.5%
bicg	0.4	0.7	5	5	23	21.7%	21.7%
cholesky	0.3	0.8	16	39	89	17.9%	43.8%
lu	0.6	1.9	15	35	77	19.4%	45.4%

Table 5.4: Summary of experiments

Experiment Set	Description
Native_FIC_EM-I	A fault-injection campaign (FIC) using error model I on the native version of a target benchmark.
Native_FIC_EM-II	Same as Native_FIC_EM-I except that error model II is used.
Presage_FIC_EM-I	A fault-injection campaign (FIC) using error model I on benchmarks transformed using PRESAGE.
Presage_FIC_EM-II	Same as Presage_FIC_EM-I except that error model II is used.

injection campaigns (FIC) where each FIC comprises 50 independent *fault injection runs*. The crash rate calculated for each *fault injection campaign* is considered as a unique random sample. Our approach is to run a sufficient number of *fault injection campaigns* until: (1) the sample distribution becomes normal or near-normal; and (2) for a target *confidence level* of 95%, the *margin of error* for the distribution falls within the range of $\pm$3.5%. We observe that for each benchmark, running 100 *fault injection campaigns* under each experiment set is sufficient to achieve a 95% confidence level with a margin of error of $\pm$3.5%.

In each *experimental run*, we carry out a *fault-free* and a *faulty* execution of a target benchmark using identical program input parameters and compare the outcome of the two executions. The program input parameters (such as array size used in the benchmark) are randomly chosen from a predefined range of values. During a *fault-free* execution, no faults are injected, whereas during a *faulty* execution, a single-bit fault is injected in a dynamic LLVM IR instruction selected randomly using either error model I or error model II as explained in Section 5.3.

Note that we only target the key function(s) that implement the core logic of a benchmark for fault injection. For example, in the `jacobi-2d` benchmark, we only target the *kernel_jacobi_2d* which implements the core Jacobi kernel and ignore the other auxiliary functions such as the function used for array initialization or the program's *main()*.

Given that the benchmarks chosen produce one or more *result arrays* as the final program output, we compare respective elements of the *result arrays* produced by the *faulty* and *fault-free* executions to categorize the outcome of the *experimental run* as:

- **SDC**: The executions ran to completion, but the corresponding elements of the *result arrays* of the *fault-free* and *faulty* execution are not equivalent.
- **Benign**: The corresponding elements of the *result arrays* of the *fault-free* and *faulty* execution are equivalent.
- **Program-crash**: The program crashes or terminates prematurely without producing the final output.

We analyze the impact of PRESAGE transformations on an application's resiliency by comparing the outcomes of the experiment sets Native_FIC_EM I with Presage_FIC_EM-I, and Native_FIC_EM-II with Presage_FIC_EM-II.

5.4.2 Fault Injection Campaigns

Figure 5.12 shows the result of FIC done under each experiment set listed in Table 5.4. Each column in the figure represents 100 FIC where each FIC consists of 50 runs. Therefore, the total number of fault injections done across 10 benchmarks and 4 experiment sets stand at 0.2 million (4 experiment sets $\times$ 10 benchmarks $\times$ 5000 fault injections).

- **Nontrivial SDC Rates**: The results for experiment sets Native_FIC_EM-I and Native_FIC_EM-II shown in Figure 5.12 demonstrate that nontrivial SDC rates are observed when *structured address computations* are subjected to bit-flips. Specifically, for the experiment set Native_FIC_EM-I, we observe a maximum and a minimum SDC rate of 32.2% and 18.5%, for the benchmarks `trmm` and `bicg`, respectively. In the case of Native_FIC_EM-II, we observe a greater contrast, with maximum SDC rate of 43.6% and a minimum SDC rate of 2.3% for the benchmarks `trmm` and `adi`, respectively.

- **Promotion of SDCs to Program-crashes**: When comparing the results of experiment sets Presage_FIC_EM-I and Presage_FIC_EM-II with that of Native_FIC_EM-I and Native_FIC_EM-II, we observe that PRESAGE transformations lead to a sizable fraction of SDCs getting promoted to program-crashes. Specifically, Presage_FIC_EM-I reports an average increase of 12.5% (averaged across all ten benchmarks) in the number of program crashes when compared to Native_FIC_EM-I, with a maximum increase of 19.3% reported for the `cholesky` benchmark. Similarly, Presage_FIC_EM-II reports an average increase of 7.8% (averaged across all ten benchmarks) in the number of program crashes when compared to Native_FIC_EM-II with a maximum increase of 16.8% reported for the `jacobi-2d` benchmark.

5.4.3 Detection Rate and Performance Overhead

Figure 5.13 shows the percentage of SDCs reported in Figure 5.12 under Presage_FIC_EM-I that are detected by the PRESAGE-inserted error detectors. Except for the benchmark `fdtd-2d`, we are able to detect 100% of the SDCs caused by a random bit-flip injected using error model I. In case of `fdtd-2d`, we are able to detect only 74% of the reported SDCs because a fraction of `GEP` instructions in `fdtd-2d` have mutable base addresses. Recall that the PRESAGE transformations can only be applied to `GEP` instructions with immutable

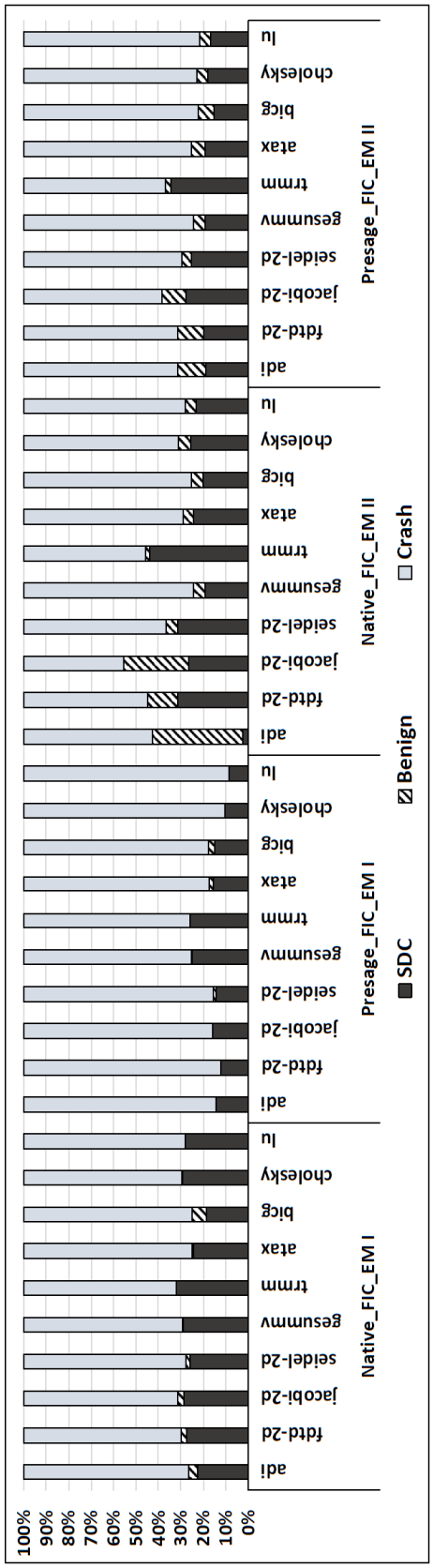

Figure 5.12: Outcomes of the fault injection campaigns

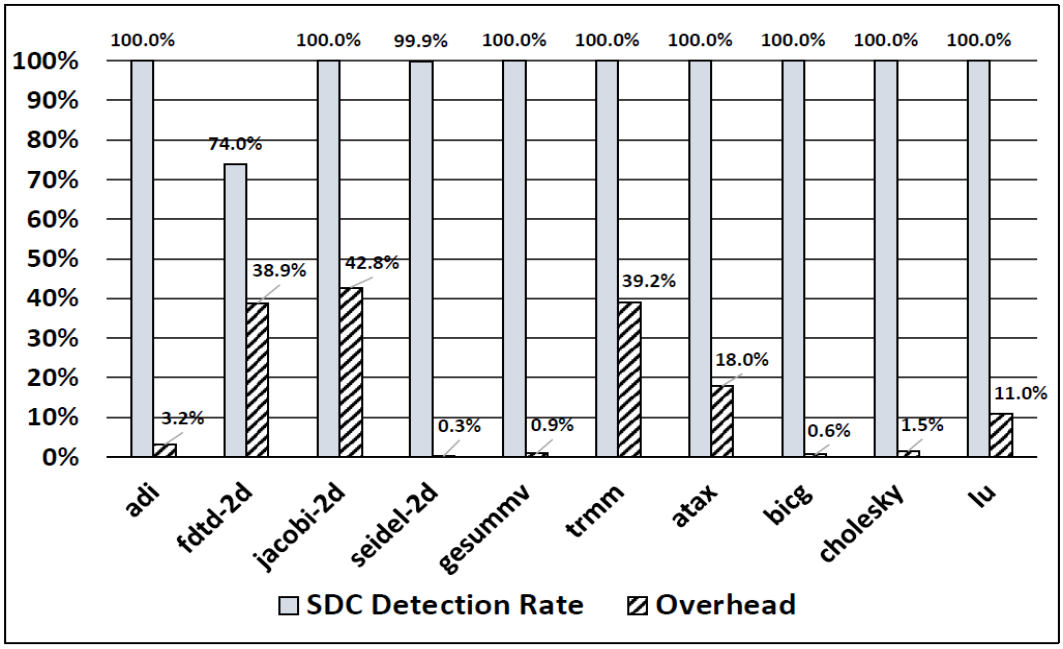

Figure 5.13: SDC detection rate and performance overhead

base addresses. For the benchmarks `adi`, `seidel-2d`, `gesummv`, `bicg`, and `cholesky`, we notice that the error detectors incur almost negligible overheads ranging between 0.3% and 3.2%. Benchmarks `lu` and `atax` report overhead figures of less than 20%, whereas the benchmarks `jacobi-2d`, `fdtd-2d`, `trmm`, and `atax` report overhead figures of close to 40%.

5.4.4 False Positives and False Negatives

We refer to the errors flagged during the execution of a PRESAGE-transformed program as a false positive when no faults are injected during the execution. Conversely, if there are no errors reported during the execution of a PRESAGE-transformed program while an error is injected during the execution, we regard it as a false negative. The basic philosophy of the PRESAGE detectors is to recompute the final address observed at the end point of a dependency-chain and compare the recomputed address against the observed final address. Also, in error model I, the index and the base address of a `GEP` instruction are assumed to be corruption-free, but the final address computed by it can be erroneous. Therefore, under error model I, whenever the recomputed address does not match the observed address, it attributes it to an actual bit-flip. In summary, the detectors never report false positives under error model I. Even in the case of error model II, where we subject the index value of a `GEP` instruction to a bit-flip, the value

recomputed by the detectors would use the same corrupted index value to reproduce the same corrupted observed value. Thus even under error model II, the error detectors must not report false positives. However, it may report false negatives, including in cases where we inject bit-flips into `GEP` instructions that have mutable base addresses, as in the case of the `fdtd-2d` benchmark.

5.4.5 Coverage Analysis

Table 5.3 provides an insight into the kind of coverage provided by the PRESAGE-based error detectors. Total SIC denotes the total static instruction count of the LLVM IR instructions corresponding to core functions that we target for fault injections in a benchmark. SIC-I and SIC-II represent the subset of instructions represented by SIC chosen using error models I and II, respectively. Apparently, SIC-I and SIC-II represent a significant portion of SIC with the share of SIC-I ranging between 15.5% and 28.5%, whereas that of SIC-II ranges between 63.3% and 21.7%. The ratio between SIC-I and SIC-II roughly varies from 1:3 (in case of `seidel-2d`) to 1:1 (in case of `gesummv` and `bicg`). Avg. DIC-I is a counterpart of SIC-I, representing the average dynamic instruction count averaged over DIC observed during each experimental run of an FIC done under the experiment set Native_FIC_EM-I. Similarly, Avg. DIC-II denotes the average dynamic instruction count averaged over DIC observed during each experimental run of an FIC done under the experiment set Native_FIC_EM-II. Clearly, the fault sites considered under error model I and II constitute a significant part of the overall static instruction count of the benchmarks considered in our experiments.

5.5 Related Work

Previous work by Casas-Guix et al. [20] shows that an Algebraic Multigrid (AMG) solver is relatively immune to faults and can, often, recover to an acceptable final answer even after encountering a momentary bit-flip in the data state. However, they realize that any fault in the space of pointers often wreaks havoc, since the corrupted pointers tend to write data values into unintended memory spaces. As a solution, they propose the use of pointer triplication, which not only helps detect errors in the value of a pointer variable but also correct the same. Unfortunately, pointer triplication comes with a high overhead of runtime checks. Also, they do not focus on the scenarios where corruptions in *structured*

address computations lead to SDC, which is the key focus of our work.

Another work by Wei et al. [110] highlights the difference between the results of the fault injection experiments done using a higher-level fault injector LLFI targeting instructions at LLVM IR-level, and a lower-level, PIN-based, fault injector performing fault injections at x86-level. This work highlights that LLVM offers a separate instruction called `getElementPtr` for carrying out *structured address computations* whereas, at x86-level, the same instruction can be used for computing address as well as performing non-address arithmetic computations. Another recent work by Nagarakatte et al. [69] shows how, by associating meta-data and by using Intel's recently introduced MPX instructions, one can guard C/C++ programs against pointer-related memory attacks. The key portion of this work is also implemented using LLVM infrastructure. The above two works, in a way, influenced our decision to choose LLVM for implementing PRESAGE.

Researchers have also explored the development of application-level error detectors for detecting soft errors affecting a program's control states [50, 72, 96]. Another critical area in application-level resilience is algorithm-based fault tolerance (ABFT), which exploits algorithmic properties of well-known applications to derive efficient error detectors [101, 102]. Researchers have also focused in the past to optimize the placement of application-level error detectors at strategic program points.The information about these strategic locations is usually derived through well-established static and dynamic program analysis techniques [32,38,75,92]. To the best of our knowledge, none of the previous works have focused on protecting *structured address generation* leading to SDC, the focus of our work.

5.6 Discussion

We have conducted preliminary investigations on the elevated overhead figures associated with some of our benchmarks. A significant portion of these overheads is attributable to the core PRESAGE transformations that introduce dependency-chains. In general, such serial dependence chains can cause: (i) increased register pressure leading to register spills, and (ii) potential loss in optimization opportunities such as vectorization. Register pressure escalations can, in general, be expected due to *structured address computations* in a basic block requiring a previously computed address from one of its predecessor basic blocks. An added side effect of such dependency-chains can be the elimination of

vectorization opportunities.

We provide here a summary of our envisaged approaches to mitigate these limitations. One approach is to split dependency-chains into shorter chains, striking a good balance between detection rates and overhead. Another approach is to create dependency-chains across instruction accesses situated a certain stride apart; this has the potential to retain a sufficient amount of exposed instruction-level parallelism while also creating address calculation chains. Last but not least, it appears worthwhile to investigate how the advantages of vectorization and dependency-chains can be obtained simultaneously.

5.7 Conclusion

Researchers in the HPC community have highlighted the growing need for developing cross-layer resilience solutions with application-level techniques gaining a prominent place due to their inherent flexibility. Developing efficient and lightweight error detectors have been a central theme of application-level resilience research dealing with silent data corruption. Through this work, we argue that, often, protecting *structured address computations* is important due to their vulnerability to bit-flips, resulting in nontrivial SDC rates. We experimentally support this argument by carrying out fault injection driven experiments on ten well-known benchmarks. We witness SDC rates ranging between 18.5% and 43.6% when instructions in these benchmarks pertaining to *structured address computations* are subjected to bit-flips.

Next, guided by the principle that maximizing the propagation of errors would make them easier to detect, we introduce a novel approach for rewriting the address computation logic used in *structured address computations*. The rewriting scheme, dubbed the RBA scheme, introduces a dependency chain in the address computation logic, enabling sufficient propagation of any error and, thus allowing efficient placement of error detectors. Another salient feature of this scheme is that it promotes a fraction of SDCs (user-invisible) to program-crashes (user-visible). One can argue that promoting SDCs to program-crashes may lead to a bad user experience. However, a program-crash is far better than an SDC, whose insidious nature does not raise any user alarms while silently invalidating the program output.

We have implemented our scheme as a compiler-level technique called PRESAGE developed using the LLVM compiler infrastructure. In Section 5.3, we formally presented

the key steps involved in implementing the PRESAGE transformations which include creating inter-block and intra-block dependency-chains, and a lightweight detector placed strategically at all exit points of a program. We reported high detection rates ranging between 74% and 100% with the performance overhead ranging between 0.3% and 42.8% across ten benchmarks. When faults are injected using error model I, the PRESAGE-transformed benchmarks witness an average and a maximum increase of 12.5% and 19.3%, respectively, in program-crashes as compared to their original versions. These figures stand at 7.8% and 16.8%, respectively, when error model II is used instead of error model I for fault injection.

Our current work identifies some challenges we plan to address as part of the future work. Specifically, we observe relatively higher detection overheads for some of the benchmarks, potentially due to increased register pressure and loss in vectorization opportunities due to the introduction of dependency-chains as explained earlier. In the future, we plan to explore efficient ways of mining `GEP` instructions in a program that are best suited for PRESAGE transformations and also vectorize the dependency-chains (wherever possible) to minimize the performance impact. Although the primary focus of our work is to provide coverage explicitly for error model I, we also observe that PRESAGE provides partial coverage for error model II by promoting a fraction of SDCs to program-crashes. As future work, we plan to explore techniques used in the context of verification and polyhedral transformations to develop comprehensive error detection mechanisms for error model II. Finally, through this work, we hope to bring to the resilience community's notice the importance and the need for developing efficient error detectors for protecting *structured address computations*.

CHAPTER 6

CONCLUDING REMARKS

In this dissertation work, we started with highlighting the need for efficient evaluation infrastructures capable of implementing various error models requiring simulation of soft errors. To this end, we developed two LLVM-level fault injectors, KULFI and VULFI, targeting scalar and vector architectures, respectively. Our developmental efforts were mainly triggered by the lack of publicly available fault injectors suiting our research requirements at that time. We also contributed to the system resilience community by making both the tools open-source. Our next key contribution focused on developing efficient error detectors capable of detecting control-flow deviations. The idea is built on the tenets of a well-known formal technique called predicate-abstraction. To make these detectors scalable, we further developed a compiler-level tool named FUSED which facilitates the automatic deployment of these control-flow detectors.

Next, driven by the curiosity of developing efficient error detectors for a class of HPC applications involving *data-intensive* computations which are not control-flow rich in nature, we carried out a case study on a 25-point RTM stencil kernel to explore the feasibility of deriving computationally less expensive approximate kernels using machine learning, to be used as redundant checkers for detecting errors in stencil computations. The study presented some key finding such as the suitability of using approximate kernels as checkers only for higher-order stencils. Our most recent work focuses on protecting structured address computation by presenting a novel relative base addressing (RBA) scheme. This scheme enables the flow of errors in *structured address computation* logic, thus helping in strategic placement of error detectors. Finally, we followed up all presented techniques in this dissertation work with an extensive set of experimental results demonstrating the effectiveness of these techniques.

To summarize, through this dissertation work, we hope to make a tiny but significant

enough contribution to the considerably vast area of system resilience. We believe the quest for developing efficient resilience solutions is far from over. With ever evolving transistor technology, new programming languages, architectures, and compiler technologies, there is an urgent need to keep a persistent effort towards developing efficient resilience solutions targeting various system layers. Looking towards exascale computing and beyond, we need a collection of resilience solutions at each system layer such that a combination of best suited cross-layer solutions can be chosen guided by the requirements of these massively parallel systems and the nature of applications running on it while keeping in mind the cost and energy implications of these resilience solutions.

REFERENCES

[1] Clang: a C Language Family Frontend for LLVM. `http://clang.llvm.org/`.

[2] GCC, the GNU Compiler Collection. `https://gcc.gnu.org/`.

[3] ISPC: Intel SPMD Program Compiler. `https://ispc.github.io/`.

[4] VULFI - An LLVM-based Fault Injection Framework. `https://fv.cs.utah.edu/fmr/vulfi`.

[5] AIKEN, L. S., WEST, S. G., AND PITTS, S. C. *Multiple Linear Regression*. John Wiley & Sons, Inc., 2003.

[6] ASHRAF, R., GIOIOSA, R., KESTOR, G., DEMARA, R. F., CHER, C., AND BOSE, P. Understanding the Propagation of Transient Errors in HPC Applications. In *International Conference for High Performance Computing,Networking, Storage and Analysis (SC)* (2015), pp. 1–12.

[7] BALL, T. A Theory of Predicate-Complete Test Coverage and Generation. In *International Conference on Formal Methods for Components and Objects (FMCO)* (2005), pp. 1–22.

[8] BALL, T., MAJUMDAR, R., MILLSTEIN, T., AND RAJAMANI, S. K. Automatic Predicate Abstraction of C Programs. In *International Conference on Programming Language Design and Implementation (PLDI)* (2001), pp. 203–213.

[9] BAYSAL, E., KOSLOFF, D. D., AND SHERWOOD, J. W. Reverse time migration. *Geophysics 48*, 11 (1983), 1514–1524.

[10] BENSON, A. R., SCHMIT, S., AND SCHREIBER, R. Silent Error Detection in Numerical Time-stepping Schemes. *International Journal of High Performance Computing Applications 29*, 4 (2015), 403–421.

[11] BERROCAL, E., BAUTISTA-GOMEZ, L., DI, S., LAN, Z., AND CAPPELLO, F. Lightweight Silent Data Corruption Detection Based on Runtime Data Analysis for HPC Applications. In *International Symposium on High-Performance Parallel and Distributed Computing (HPDC)* (2015), pp. 275–278.

[12] BIENIA, C. *Benchmarking Modern Multiprocessors*. PhD thesis, Princeton University, 2011.

[13] BIENIA, C., KUMAR, S., SINGH, J. P., AND LI, K. The PARSEC Benchmark Suite: Characterization and Architectural Implications. In *International Conference on Parallel Architectures and Compilation Techniques (PACT)* (2008), pp. 72–81.

[14] BORIN, E., WANG, C., WU, Y., AND ARAUJO, G. Dynamic Binary Control-flow

Errors Detection. *Computer Architecture News* (2005), 15–20.

[15] BORKAR, S. Design Challenges of Technology Scaling. *IEEE Micro 19*, 4 (1999), 23–29.

[16] BRADLEY, A. P. The Use of the Area Under the ROC Curve in the Evaluation of Machine Learning Algorithms. *Pattern Recognition 30*, 7 (1997), 1145–1159.

[17] BRONEVETSKY, G. *Portable Checkpointing for Parallel Applications*. PhD thesis, 2007.

[18] CAPPELLO, F., GEIST, A., GROPP, B., KALE, L., KRAMER, B., AND SNIR, M. Toward Exa-scale Resilience. *International Journal of High Performance Computing Applications 23*, 4 (2009), 374–388.

[19] CAPPELLO, F., GEIST, A., GROPP, W., KALE, S., KRAMER, B., AND SNIR, M. Toward Exascale Resilience: 2014 Update. *Supercomputing frontiers and innovations 1*, 1 (2014), 5–28.

[20] CASAS, M., DE SUPINSKI, B. R., BRONEVETSKY, G., AND SCHULZ, M. Fault Resilience of the Algebraic Multi-Grid Solver. In *International Conference on Supercomputing (ICS)* (2012), pp. 91–100.

[21] CEBRIAN, J., JAHRE, M., AND NATVIG, L. Optimized Hardware for Suboptimal Software: The Case for SIMD-aware Benchmarks. In *International Symposium on Performance Analysis of Systems and Software (ISPASS)* (2014), pp. 66–75.

[22] CHANG, C.-C., AND LIN, C.-J. LIBSVM: A library for support vector machines. *ACM Transactions on Intelligent Systems and Technology 2* (2011), 1–27. Software available at `http://www.csie.ntu.edu.tw/~cjlin/libsvm`.

[23] CHEN, C., AND HSIAO, M. Error-Correcting Codes for Semiconductor Memory Applications: A State-of-the-Art Review. *IBM Journal of Research and Development 28*, 2 (1984), 124–134.

[24] CHEN, S. personal communication, 2013.

[25] CHENG, E., MIRKHANI, S., SZAFARYN, L. G., CHER, C., CHO, H., SKADRON, K., STAN, M. R., LILJA, K., ABRAHAM, J. A., BOSE, P., AND MITRA, S. CLEAR: Cross-Layer Exploration for Architecting Resilience - Combining Hardware and Software Techniques to Tolerate Soft Errors in Processor Cores. In *Design Automation Conference (DAC)* (2016), p. 68.

[26] CHER, C., GUPTA, M. S., BOSE, P., AND MULLER, K. P. Understanding Soft Error Resiliency of Blue Gene/Q Compute Chip through Hardware Proton Irradiation and Software Fault Injection. In *International Conference for High Performance Computing, Networking, Storage and Analysis (SC)* (2014), pp. 587–596.

[27] CORTES, C., AND VAPNIK, V. Support-Vector Networks. *Machine Learning 20*, 3 (1995), 273–297.

[28] DAS, S., DILL, D., AND PARK, S. Experience with Predicate Abstraction. In *International Conference on Computer Aided Verification (CAV)*. 1999, pp. 160–171.

[29] DAVIES, T., AND CHEN, Z. Correcting Soft Errors Online in LU Factorization. In *International Symposium on High-performance Parallel and Distributed Computing (HPDC)* (2013), pp. 167–178.

[30] DAVIS, T. A., AND HU, Y. The University of Florida Sparse Matrix Collection. *in ACM Transactions on Mathematical Software (TOMS) 38*, 1 (2011), 1–25.

[31] DE KRUIJF, M., NOMURA, S., AND SANKARALINGAM, K. Relax: An Architectural Framework for Software Recovery of Hardware Faults. In *International Symposium on Computer Architecture (ISCA)* (2010), pp. 497–508.

[32] DE KRUIJF, M. A., SANKARALINGAM, K., AND JHA, S. Static Analysis and Compiler Design for Idempotent Processing. *International Conference on Programming Language Design and Implementation (PLDI)* (2012), 475–486.

[33] DEMMEL, J. W., EISENSTAT, S. C., GILBERT, J. R., LI, X. S., AND LIU, J. W. H. A Supernodal Approach to Sparse Partial Pivoting. 720–755.

[34] DING, C., KARLSSON, C., LIU, H., DAVIES, T., AND CHEN, Z. Matrix Multiplication on GPUs with On-Line Fault Tolerance. In *International Symposium on Parallel and Distributed Processing with Applications (ISPA)* (2011), pp. 311–317.

[35] DU, P., BOUTEILLER, A., BOSILCA, G., HERAULT, T., AND DONGARRA, J. Algorithm-based Fault Tolerance for Dense Matrix Factorizations. In *International Symposium on Principles and Practice of Parallel Programming (PPoPP)* (2012), pp. 225–234.

[36] FANG, B., PATTABIRAMAN, K., RIPEANU, M., AND GURUMURTHI, S. Evaluating the Error Resilience of Parallel Programs. In *International Conference on Dependable Systems and Networks (DSN)* (2014), pp. 720–725.

[37] FANG, B., PATTABIRAMAN, K., RIPEANU, M., AND GURUMURTHI, S. GPU-Qin : A Methodology for Evaluating the Error Resilience of GPGPU Applications. In *International Symposium on Performance Analysis of Systems and Software (ISPASS)* (2014), pp. 221–230.

[38] FENG, S., GUPTA, S., ANSARI, A., AND MAHLKE, S. Shoestring: Probabilistic Soft Error Reliability on the Cheap. In *International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)* (2010), pp. 385–396.

[39] FERRARO-PETRILLO, U., FINOCCHI, I., AND ITALIANO, G. F. The Price of Resiliency: a Case Study on Sorting with Memory Faults. *Algorithmica 53*, 4 (2009), 597–620.

[40] FERRARO-PETRILLO, U., FINOCCHI, I., AND ITALIANO, G. F. Experimental study of resilient algorithms and data structures. In *International Conference on Experimental Algorithms (SEA)* (2010), pp. 1–12.

[41] Open Source Scientific Computing Library. `http://people.sc.fsu.edu/~jburkardt/`.

[42] GRAF, S., AND SAÏDI, H. Construction of abstract state graphs with PVS. In

International Conference on Computer Aided Verification (CAV) (1997), pp. 72–83.

[43] HARI, S. K. S., ADVE, S. V., NAEIMI, H., AND RAMACHANDRAN, P. Relyzer: Exploiting application-level fault equivalence to analyze application resiliency to transient faults. In *International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)* (2012), pp. 123–134.

[44] HARI, S. K. S., TSAI, T., STEPHENSON, M., KECKLER, S. W., AND EMER, J. SASSIFI: Evaluating Resilience of GPU Applications. In *Silicon Errors in Logic – System Effects (SELSE)* (2015).

[45] HEIDEL, D. F., RODBELL, K. P., CANNON, E. H., CABRAL, C., GORDON, M. S., OLDIGES, P., AND TANG, H. H. Alpha-particle-induced Upsets in Advanced CMOS Circuits and Technology. *IBM Journal of Research and Development 52*, 3 (2008), 225–232.

[46] HSUEH, M.-C., TSAI, T., AND IYER, R. Fault Injection Techniques and Tools. *Computer 30*, 4 (1997), 75–82.

[47] HUKERIKAR, S., DINIZ, P. C., AND LUCAS, R. F. Programming Model Extensions for Resilience in Extreme Scale Computing. In *Euro-Par Parallel Processing Workshops*. (2012), pp. 496–498.

[48] JIN, A., JIANG, J., HU, J., AND LOU, J. A PIN-based dynamic software fault injection system. In *International Conference for Young Computer Scientists (ICYCS)* (2008), pp. 2160–2167.

[49] KANAWATI, G., KANAWATI, N., AND ABRAHAM, J. FERRARI: A Flexible Software-based Fault and Error Injection System. *IEEE Transactions on Computers 44*, 2 (1995), 248–260.

[50] KHUDIA, D. S., AND MAHLKE, S. A. Low Cost Control-flow Protection Using Abstract Control Signatures. In *Languages, Compilers and Tools forEmbedded Systems (LCTES)* (2013), pp. 3–12.

[51] KIM, J., SULLIVAN, M., AND EREZ, M. Bamboo ECC: Strong, Safe, and Flexible Codes for Reliable Computer Memory. In *International Symposium on High Performance Computer Architecture (HPCA)* (2015), pp. 101–112.

[52] KIM, J., SULLIVAN, M., GONG, S., AND EREZ, M. Frugal ECC: Efficient and Versatile Memory Error Protection through Fine-grained Compression. In *International Conference for High Performance Computing, Networking, Storage and Analysis (SC)* (2015), pp. 1–12.

[53] KOHAVI, R. A Study of Cross-validation and Bootstrap for Accuracy Estimation and Model Selection. In *International Joint Conference on Artificial Intelligence (IJCAI)* (1995), pp. 1137–1143.

[54] KULFI: An Instruction Level Fault Injector. `https://github.com/soarlab/KULFI`.

[55] LATTNER, C., AND ADVE, V. LLVM: A Compilation Framework for Lifelong Program Analysis and Transformation. In *International Symposium on Code Generation*

and Optimization (CGO) (2004), pp. 75–86.

[56] LI, J., AND DRAPER, J. Accelerating Soft error-rate (SER) Estimation in the Presence of Single Event Transients. In *Design Automation Conference (DAC)* (2016), pp. 1–6.

[57] LI, X., DEMMEL, J., GILBERT, J., IL. GRIGORI, SHAO, M., AND YAMAZAKI, I. SuperLU Users' Guide. Tech. Rep. LBNL-44289, Lawrence Berkeley National Laboratory, 1999.

[58] The LLVM Compiler Infrastructure. `http://llvm.org/`.

[59] LU, Q., FARAHANI, M., WEI, J., THOMAS, A., AND PATTABIRAMAN, K. LLFI: An Intermediate Code-Level Fault Injection Tool for Hardware Faults. In *International Conference on Software Quality, Reliability and Security (QRS)* (2015), pp. 11–16.

[60] LYLE, G., CHEN, S., PATTABIRAMAN, K., KALBARCZYK, Z., AND IYER, R. An End-to-End Approach for the Automatic Derivation of Application-aware Error Detectors. In *International Conference on Dependable Systems and Networks (DSN)* (2009), pp. 584–589.

[61] MADEIRA, H., RELA, M. Z., MOREIRA, F., AND SILVA, J. A. G. RIFLE: A General Purpose Pin-level Fault Injector. In *International Symposium on Dependable Computing (PRDC)*. 1994, pp. 197–216.

[62] MAY, T. C., AND WOODS, M. H. Alpha-particle-induced Soft Errors in Dynamic Memories. *IEEE Transactions on Electron Devices 26*, 1 (1979), 2–9.

[63] MCCOOL, M., ROBISON, A. D., AND REINDERS, J. Chapter 10: Forward Seismic Simulation. In *Structured Parallel Programming: Patterns for Efficient Computation*, 1st edition ed. 2012, pp. 265–277.

[64] MEANEY, P., SWANEY, S., SANDA, P., AND SPAINHOWER, L. IBM z990 Soft Error Detection and Recovery. *IEEE Transactions on Device and Materials Reliability 5*, 3 (2005), 419–427.

[65] MICHALAK, S., HARRIS, K., HENGARTNER, N., TAKALA, B., AND WENDER, S. Predicting the Number of Fatal Soft Errors in Los Alamos National Laboratory's ASC Q Supercomputer. *IEEE Transactions on Device and Materials Reliability (DMR) 5*, 3 (2005), 329–335.

[66] MITRA, S., SEIFERT, N., ZHANG, M., SHI, Q., AND KIM, K. S. Robust System Design with Built-In Soft-Error Resilience. *Computer 38* (2005), 43–52.

[67] MUKHERJEE, S. *Architecture Design for Soft Errors*. Morgan Kaufmann Publishers Inc., 2008.

[68] MUKHERJEE, S. S., WEAVER, C., EMER, J., REINHARDT, S. K., AND AUSTIN, T. A Systematic Methodology to Compute the Architectural Vulnerability Factors for a High-Performance Microprocessor. In *Microarchitecture (MICRO)* (2003), p. 29.

[69] NAGARAKATTE, S., MARTIN, M. M., AND ZDANCEWIC, S. Everything You Want to Know About Pointer-Based Checking. In *Summit on Advances in Programming*

Languages (SNAPL) (2015), pp. 190–208.

[70] NEYMAN, J. On the Two Different Aspects of the Representative Method: The Method of Stratified Sampling and the Method of Purposive Selection. *Journal of the Royal Statistical Society 97*, 4 (1934), 558–625.

[71] NOH, J., CORREAS, V., LEE, S., JEON, J., NOFAL, I., CERBA, J., BELHADDAD, H., ALEXANDRESCU, D., LEE, Y., AND KWON, S. Study of Neutron Soft Error Rate (SER) Sensitivity: Investigation of Upset Mechanisms by Comparative Simulation of FinFET and Planar MOSFET SRAMs. *IEEE Transactions on Nuclear Science 62*, 4 (2015), 1642–1649.

[72] OH, N., SHIRVANI, P. P., AND MCCLUSKEY, E. J. Control-Flow Checking by Software Signatures. *IEEE Transactions on Reliability 51*, 1 (2002), 111–122.

[73] The open standard for parallel programming of heterogeneous systems. `https://www.khronos.org/opencl/`.

[74] PARK, C. N., AND DUDYCHA, A. L. A Cross-validation Approach to Sample Size Determination for Regression Models. *Journal of the American Statistical Association 69*, 345 (1974), 214–218.

[75] PATTABIRAMAN, K., KALBARCZYK, Z., AND IYER, R. Application-Based Metrics for Strategic Placement of Detectors. In *International Symposium on Pacific Rim Dependable Computing (PRDC)* (2005), pp. 75–82.

[76] PATTABIRAMAN, K., KALBARCZYK, Z., AND IYER, R. Automated Derivation of Application-Aware Error Detectors Using Static Analysis: The Trusted Illiac Approach. *IEEE Transactions on Dependable and Secure Computing 8*, 1 (2011), 44–57.

[77] PATTABIRAMAN, K., KALBARCZYK, Z., AND IYER, R. K. Automated Derivation of Application-aware Error Detectors using Static Analysis. In *International On-Line Testing Symposium (IOLTS)* (2007), pp. 211–216.

[78] PATTABIRAMAN, K., NAKKA, N., KALBARCZYK, Z., AND IYER, R. SymPLFIED: Symbolic Program-level Fault Injection and Error Detection Framework. In *IEEE International Conference on Dependable Systems and Networks (DSN)* (2008), pp. 472–481.

[79] PATTABIRAMAN, K., SAGGESE, G., CHEN, D., KALBARCZYK, Z., AND IYER, R. Automated Derivation of Application-Specific Error Detectors Using Dynamic Analysis. *in IEEE Transactions on Dependable and Secure Computing 8*, 5 (2011), 640–655.

[80] PERRY, F., AND WALKER, D. Reasoning about Control-Flow in the Presence of Transient Faults. In *Static Analysis Symposium (SAS)*. 2008, pp. 332–346.

[81] PHARR, M., AND MARK, W. R. ISPC: A SPMD Compiler for High-performance CPU Programming. In *Innovative Parallel Computing (InPar)* (2012), pp. 1–13.

[82] Polybench benchmark suite. `https://sourceforge.net/projects/polybench`.

[83] PolyBench/C: The polyhedral benchmark suite. `http://web.cs.ucla.edu/`

~pouchet/software/polybench/.

[84] QUINLAN, D. J. Rose: Compiler support for object-oriented frameworks. *in Parallel Processing Letters (PPL) 10*, 2/3 (2000), 215–226.

[85] RACUNAS, P., CONSTANTINIDES, K., MANNE, S., AND MUKHERJEE, S. Perturbation-based Fault Screening. In *International Symposium on High Performance Computer Architecture (HPCA)* (2007), pp. 169–180.

[86] RIVERS, J., AND KUDVA, P. Reliability Challenges and System Performance at the Architecture Level. *in IEEE Design Test of Computers (DTC) 26*, 6 (2009), 62–73.

[87] RIVERS, J. A., GUPTA, M. S., SHIN, J., KUDVA, P. N., AND BOSE, P. Error Tolerance in Server Class Processors. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems 30*, 7 (2011), 945–959.

[88] ROSE: A Compiler Infrastructure. `http://rosecompiler.org/`.

[89] S. BORKAR. Designing Reliable Systems from Unreliable Components: the Challenges of Transistor Variability and Degradation. *IEEE Micro 25*, 6 (2005), 10–16.

[90] SAHOO, S. K., LI, M.-L., RAMACHANDRAN, P., V.ADVE, S., ADVE, V. S., AND ZHOU, Y. Using Likely Program Invariant to detect Hardware Errors. In *International Conference on Dependable Systems and Networks (DSN)* (2008), pp. 70–79.

[91] SANDA, P. N., KELLINGTON, J. W., KUDVA, P., KALLA, R. N., MCBETH, R. B., ACKARET, J., LOCKWOOD, R., SCHUMANN, J., AND JONES, C. R. Soft Error Resilience of the IBM POWER6 Processor. *IBM Journal of Research and Development 52*, 3 (2008), 275–284.

[92] SASTRY HARI, S., ADVE, S., NAEIMI, H., AND RAMACHANDRAN, P. Relyzer: Application Resiliency Analyzer for Transient Faults. *IEEE MICRO 33*, 3 (2013), 58–66.

[93] SHARMA, V. C., BRONEVETSKY, G., AND GOPALAKRISHNAN, G. Detecting Soft Errors in Stencil based Computations. In *IEEE Workshop on Silicon Errors in Logic—System Effects (SELSE)* (2015). Poster paper.

[94] SHARMA, V. C., GOPALAKRISHNAN, G., AND KRISHNAMOORTHY, S. PRESAGE: Protecting Structured Address Generation against Soft Errors. In *International Conference on High Performance Computing (HiPC)* (2016), pp. 252–261.

[95] SHARMA, V. C., GOPALAKRISHNAN, G., AND KRISHNAMOORTHY, S. Towards Resiliency Evaluation of Vector Programs. In *International Parallel and Distributed Processing Symposium Workshops (DPDNS)* (2016), pp. 1319–1328.

[96] SHARMA, V. C., HARAN, A., RAKAMARIĆ, Z., AND GOPALAKRISHNAN, G. Towards Formal Approaches to System Resilience. In *International Symposium on Pacific Rim Dependable Computing (PRDC)* (2013), pp. 41–50.

[97] SHARMA, V. C., HARAN, A., RAKAMARIĆ, Z., AND GOPALAKRISHNAN, G. Towards Formal Approaches to System Resilience. In *International Symposium on*

Pacific Rim Dependable Computing (PRDC) (2013), pp. 41–50.

[98] SHARMA, V. C., RAKAMARIĆ, Z., AND GOPALAKRISHNAN, G. FUSED: A Low-cost Online Soft-Error Detector. In *IEEE Workshop on Silicon Errors in Logic—System Effects (SELSE)* (2014). Poster paper.

[99] SIEH, V. Fault-injector using UNIX ptrace interface. In *Internal Report 11/93, IMMD3, Universität Erlangen Nürnberg* (1993).

[100] SLAYMAN, C. Cache and Memory Error Detection, Correction, and Reduction Techniques for Terrestrial Servers and Workstations. *IEEE Transactions on Device and Materials Reliability 5*, 3 (2005), 397–404.

[101] SLOAN, J., KUMAR, R., AND BRONEVETSKY, G. An Algorithmic Approach to Error Localization and Partial Recomputation for Low-Overhead Fault Tolerance. In *International Conference on Dependable Systems and Networks (DSN)* (2013), pp. 1–12.

[102] TAO, D., SONG, S. L., KRISHNAMOORTHY, S., WU, P., LIANG, X., ZHANG, E. Z., KERBYSON, D., AND CHEN, Z. New-Sum: A Novel Online ABFT Scheme For General Iterative Methods. In *International Symposium on High-Performance Parallel and Distributed Computing (HPDC)* (2016), pp. 43–55.

[103] THOMAS, A., AND PATTABIRAMAN, K. Error Detector Placement for Soft Computation. In *International Conference on Dependable Systems and Networks (DSN)* (2013), pp. 1–12.

[104] VAPNIK, V. An overview of statistical learning theory. *IEEE Transactions on Neural Networks 10*, 5 (1999), 988–999.

[105] VAPNIK, V. N. The nature of statistical learning theory, 1995.

[106] VEMU, R., AND ABRAHAM, J. CEDA: Control-Flow Error Detection Using Assertions. *IEEE Transactions on Computers 60*, 9 (2011), 1233–1245.

[107] VENKATASUBRAMANIAN, R., HAYES, J. P., AND MURRAY, B. T. Low-cost On-line Fault Detection Using Control-flow Assertions. In *International On-Line Testing Symposium (IOLTS)* (2003), pp. 137–143.

[108] WANG, N., AND PATEL, S. ReStore: Symptom based Soft Error Detection in Microprocessors. In *International Conference on Dependable Systems and Networks (DSN)* (2005), pp. 30–39.

[109] WEI, J., AND PATTABIRAMAN, K. BLOCKWATCH: Leveraging Similarity in Parallel Programs for Error Detection. In *International Conference on Dependable Systems and Networks (DSN)* (2012), pp. 1–12.

[110] WEI, J., THOMAS, A., LI, G., AND PATTABIRAMAN, K. Quantifying the Accuracy of High-Level Fault Injection Techniques for Hardware Faults. In *International Conference on Dependable Systems and Networks (DSN)* (2014), pp. 375–382.

[111] WEISS, N. A. *Elementary Statistics*, 8th ed. Pearson, 2011.

[112] YIM, K. S., PHAM, C., SALEHEEN, M., KALBARCZYK, Z., AND IYER, R. Hauberk:

Lightweight Silent Data Corruption Error Detector for GPGPU. In *International Parallel and Distributed Processing Symposium (IPDPS)* (2011), pp. 287–300.

[113] YU, J., GARZARAN, M., AND SNIR, M. Efficient Software Checking for Fault Tolerance. In *International Parallel and Distributed Processing Symposium (IPDPS)* (2008), pp. 1–5.